PharmaSUG 2023 - Paper SD-255

Using Bundles for R Package Management

Magnus Mengelbier, Limelogic AB

ABSTRACT

The broader use of R within Life Sciences from a niche personal statistical analysis tool to an organization wide environment is making R package management a more complex and comprehensive task. As the organization grows and the types of analysis are ever expanded, a substantial increase in the number of packages is certain to follow. The use of R packages for dependency management, such as packrat and renv, is very applicable for personal R environments or where users are permitted to fully customize their collection of packages, but not necessarily validated GCP environments. The R bundle approach discussed is designed to provide a similar high degree of flexibility for dependency management while staying within the constraints imposed by GCP compliance. Additional benefits, such as significant decrease in complexity and improved overall life cycle of the R environment, are further explored through examples.

INTRODUCTION

R package management is a task that all R users are faced with at some time in their use of R. The task grows in complexity as the number of packages increase, more so if the packages are installed in the central library accessible to other users, such as the R application or site library.

The effort is further affected by additional business requirements and constraints to comply with an organization's Good Clinical Practice (GCP), or other wider GxP, quality and compliance standards.

The principle of a bundle is extended to R, giving a simple means to manage multiple libraries concurrently with minimizing disruption, retain the flexibility of smaller curated package collections and while at the same time simplifying validation.

The result is a simple use of directory structures and attributes that relies on standard R functionality to such a degree that it can be implemented with the minimal set of packages required by R to function. As we shall discuss, creating a bundle uses the same mechanisms, tools and functions that are used to install packages, creating an architecture that can be used within existing infrastructure and processes.

R AND LIBRARIES

R is most often described an object-oriented language for statistical analysis and graphics. A full discussion of R is beyond the scope of this paper, but there are key aspects of the R language to highlight that directly impacts or are addressed with the bundle concept, whether R is used in an interactive analysis environment, embedded or part of a larger compute cluster.

A key construct of R is that a collection of re-usable objects, methods and functions are distributed as part of an R package. A collection of R packages is usually referred to as an R package library, or simply R library. In simple practical terms, an R library is the directory where R packages are installed. These directories can either be a central common area accessible to all users or a personal and private user library that is only accessible be each individual user.

R searches for installed packages in a library tree, i.e. the directory paths defined and returned by `.libPaths()`, and in that returned order. If a package is installed in more than one location in this list of paths, R simply uses the first occurrence of the package it finds, regardless of if there is a more recent or applicable version of the package installed further down the list.

Figure 1 Common install locations for R packages

A common approach for GCP compliant R environments is to use a library tree with three principal locations (Figure 1). The minimum set of R packages that are included with and required for the R application to function is installed in the R *application library* (path represented by the R object `.Library`).

Depending on the validation approach, this may exclude recommended packages as they are optional, i.e. not required for R to function. An additional argument to exclude these packages is that many of them are still under development and/or maintenance, so updates are periodically released independently of new R versions.

The preferred approach is for any additional or add-on R packages to be installed in the R application *site library* (path represented by the R object `.Library.site`) to physically separate packages required by R to function from those that represent the curated selection of R packages that make up an organization's compute and analysis capabilities.

A third location is essentially any other directory that is accessible from within the R application. This can either be the user library, if enabled, or some directory accessible to all or a predefined group of users with the appropriate file system permission.

R does impose a certain order to the library tree, which is important to note. If not explicitly specified, the R application will append the site library (if it exists) and the application library to the library tree, and in that order, meaning that packages in paths prior to either will take precedence. This can either be an issue or desired effect if you need to explicitly control which installed packages are used.

One other important note is that the R application library and site library are by default accessible to all users, meaning any package updates will impact all users. Both the complexity and disruption significantly increase as the number of packages grow given packages natural evolving, and sometimes intricate, dependencies. Package managers like *renv* and *packrat* circumvents this by overriding the R configuration with a user's personal application library and site library, which simply means any previously installed packages are hidden from the user and not directly accessible. It also means that each *renv* or *packrat* project will need to install each package individually again, unless a central cache is employed, both which retains some of the compliance implications discussed further below.

There are a large number of different strategies to manage the R libraries above, from commercial products, package managers like *renv* and *packrat*, or simply doing it yourself, but in the end, each will still manage the application, site, user and/or custom libraries since this is how R functions. Bundles is no less different.

GCP COMPLIANCE

The question if R requires to be GCP compliant and validated is a well-discussed topic and falls within each individual organizations assessment. As a broad statement, we assume that an organization has determined that R will need some level of validation to be GCP compliant, which most probably also includes R packages. The topic of R validation in general is quite broad, but package management plays an integral role in maintaining system compliance.

A common risk-based approach is to employ Good Automated Manufacturing Practice (GAMP) as a guide or tool to determine the level of compliance and validation that software is expected to meet with respect to GCP, or any other domains of GxP for that matter.

R, as a mathematical and/or statistical language, could fall within GAMP v5 Category 1, a category that within most Life Science organization would only require that the install and continuous maintenance is documented per the organization's IT Change Request/Control standards. However, the community would most likely argue that R capabilities are defined by and provided through the R packages you install.

The R application is therefore often categorized as modular software (GAMP v5 Category 4), in large part due to that packages can enhance, re-define, adapt, alter, or modify R functionality, and that includes standard functions.

This would also encompass approaches where a user is permitted to install packages from a pre-defined approved list or controlled repository, since the user can alter the capabilities and functionality of R based on the user's package selection. Modularity is defined by how R is used and not which packages are available and from what source.

Also note that packages evolve over time with new features, functionality and fixes, which would imply that R capabilities are further controlled though which package versions you elect or decide not to install, e.g. you can selectively adapt R by selecting specific versions of a package. The modularity is further enforced if users are permitted to install packages to customize capabilities for specific projects, studies or analysis, potentially resulting in a plethora of different packages and respective versions.

An organization could also require that the modular aspect of the R environment is validated to demonstrate that the install sequence and different package combinations would not adversely impact the functionality of R given the ever-increasing permutations and complex chain of dependencies.

One additional strength of R to consider in light of compliance, is that R can utilize business or compute logic written in languages other than R, a simple example being the R package survival (Figure 2), or provided through dynamically linked shared object or loaded libraries, such as the commonly referred to ".so"-files or DLL's on Linux and Microsoft Windows environments, respectively.

```
* installing *source* package 'survival' ...
** package 'survival' successfully unpacked and MD5 sums checked
** using staged installation
** libs
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agexact.c -o agexact.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agfit4.c -o agfit4.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agfit5.c -o agfit5.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agmart.c -o agmart.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agmart3.c -o agmart3.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agscore2.c -o agscore2.o
gcc -I"/opt/R/4.0.3/lib64/R/include" -DNDEBUG   -I/usr/local/include   -fpic  -g -O2  -c agsurv4.c -o agsurv4.o
```

Figure 2 Installing R package survival

This introduces a tool chain dependency on utilities, such as the GNU c/c++ compiler (gcc above), and shared/dynamic libraries not part of the R distribution, which also need to be considered as part pre-requisite and dependency when packages are validated and installed. As those dependencies are most likely updated independently of the R application itself, say during IT security patching, tool chain dependencies need to be adequately managed as part of the package validation and management process, especially if the underlying host environment is frequently updated and users are permitted to install packages.

If, on the other hand, users are not permitted to install packages and said packages are installed in the R environment and validated prior to use, then the R environment would have a pre-defined set of capabilities, meaning R would be non-configurable software (GAMP v5 Category 3), which is a historically common and well-known software category within Life Sciences. For completeness, the only exception would be internally developed packages (GAMP v5 Category 5) or any packages that are truly modular by design (GAMP v5 Category 4).

As most packages only use utilities or statically link to the shared or dynamic libraries during install, there is also less likelihood of any impact or validation becoming void due to system updates, simply because the packages are installed once and not each time a project is initiated.

This latter category is however a static set of capabilities without the benefits of the modularity of R. As the number of packages grow, only validating updates will become an increasingly arduous task and re-validating the entire R environment may become too costly. Using bundles for package management is an approach to retain the more straightforward validation process for non-configurable software while at the same time retain some of the benefits of the modularity of R. As we shall discuss, bundles can also keep validation to within a reasonable effort.

BUNDLES

The bundle concept has been around software architecture for quite some time and the use with R, and likewise other analysis languages like SAS and Python, is based on the argument that smaller libraries of packages is far easier to maintain and validate than a single large library, such as the site library.

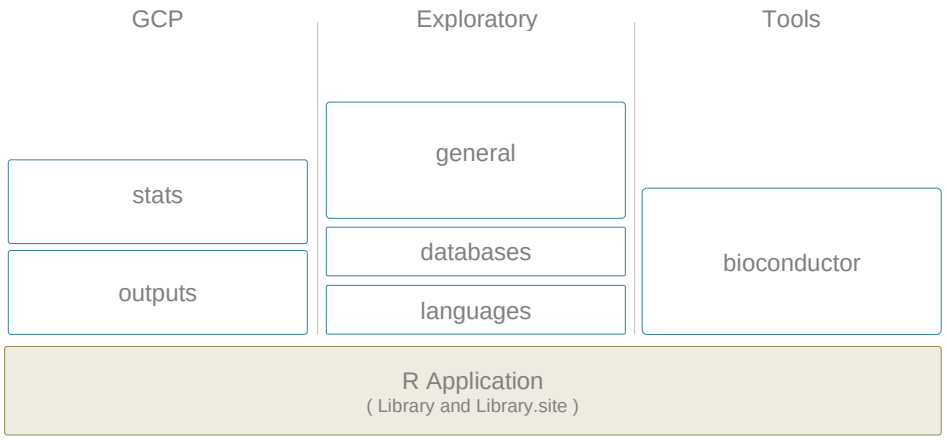

Figure 3 Purpose-built bundles

The use of bundles also provides some key benefits. Different bundles can be constructed for a particular purpose and compliance requirement (Figure 3), retaining the modular philosophy of package managers while simplifying the package maintenance, validation and compliance of pre-installed libraries. This narrower scope of use would also allow a substantially narrower window of validation as packages are not validated for general broad use, further simplifying compliance.

As a secondary benefit, the ability to version bundles would also limit disruption and significantly improve reproducibility.

Disruption to the business process as packages and their dependencies are updated can be nearly eliminated if it is permitted that a study or project can continue to use a bundle version that has previously been validated. Any updated bundles would then be released as new versions, not impacting what has previously been released.

Reproducibility would also be a simple exercise. Instead of attempting to recreate a package library in some alternative location, the study or project teams would simply just refer to the existing bundle version, with the understanding that bundle versions are not deleted unless explicitly required. Reproducibility can be further simplified if at certain study or project milestones, business processes mandate that the analysis locks to specific bundle versions. That could potentially eliminate any need for analysis programs to be updated to use a specific version (see use of explicit bundle versions below).

ARCHITECTURE

The bundle architecture is based on a simple use of directory structures, property files and standard R functionality. In essence, the bundle concept can by design be implemented with the minimum set of packages required by the R application.

The actual parent directories for bundles, and nothing restricts multiple locations, is any readable location accessible from the user's R session, our third location discussed previously. In the majority of cases, the bundle parent directories are located outside of the R application install directory to simplify the validation process and provide support for languages beyond R.

One case where bundles are installed with the R application is when R is either distributed as a virtualized application or embedded such that security constraints and restrictions does not permit access to directories external to the application itself.

For our examples, we shall use `/opt/bundles` as the parent directory for bundles, which is functionally equivalent to say a Windows path `X:\bundles`, where the X-drive can be any suitable drive.

NAME AND VERSION

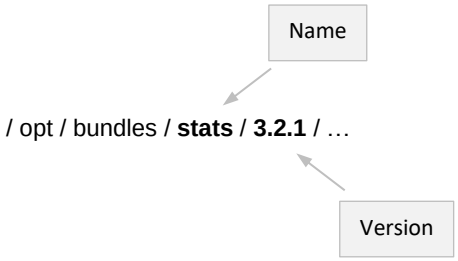

Figure 4 Bundle directories

A bundle is defined by a two-level directory structure (Figure 4), the first being the bundle name and the second the bundle version, allowing bundles to be defined and maintained independent to any other bundles. The version example above is based on semantic version conventions, but any scheme could work as long as the version reference is a valid directory name.

BUNDLE PROPERTIES

It quickly became apparent that associating certain properties to a bundle or a specific bundle version would be required. This could include defining dependencies, scope of use, such as GCP versus non-GCP, or other applicable attributes.

The mechanism of choice is a simple properties file, say the file `r.properties`, that would define properties as key/value pairs. Given the location of the properties file in either the bundle main directory or a specific version, the properties would either apply to all versions of a bundle or specific to that version, respectively.

For example, bundle dependencies, such a dependency on another bundle or external resource, is commonly defined for each bundle version due to validation constraints. Other properties, both general and version specific, could include the default bundle version, supported versions of R, workflow states, search paths, disabling user libraries, etc.

INTERNAL DIRECTORY STRUCTURES

Bundles also has an internal directory structure (Figure 5) that has evolved over time to accommodate different use cases and resolve issues, such as tool chain dependencies. It is not uncommon for a bundle version to be available for multiple R versions over time with tool chain updates between each release.

Figure 5 Internal directory structure

A bundle may contain more than just language libraries, such as shell scripts, command line tools, etc., so the `libraries` directory may be joined by `bin`, `tools`, `apps`, etc. In the example above, we concentrate on the bundle R language library.

The `R` directory allows us to also include multiple languages, such as SAS or Python. It is not uncommon that validated Python packages are included in a bundle that includes the R package *reticulate*. Or include SAS macros to permit a bundle to be used interchangeably between R and SAS:

The R version directory also provides natural support for multiple tool chains, incompatible R versions, and, if applicable, compliance constraints. R packages are usually validated for a specific version of R and, depending on your organization's quality approach, this may either be each installed R version or possibly for each minor release.

USING BUNDLES

We have discussed the concept and the architecture of bundles and only briefly mentioned discovering and accessing R packages through the library tree. In fact, bundles (Figure 6) is all about managing `.libPaths()` with user-friendly functions regardless of the environment or organization.

```
> .libPaths()
[1] "/home/user/R/x86_64-pc-linux-gnu-library/4.0" "/opt/R/4.0.3/lib64/R/site-library"
[3] "/opt/R/4.0.3/lib64/R/library"
>
> cxlib_bundle_load( "outputs/1.1.1" )
>
> .libPaths()
[1] "/home/user/R/x86_64-pc-linux-gnu-library/4.0" "/opt/bundles/outputs/1.1.1/libraries/R/4.0.3"
[3] "/opt/R/4.0.3/lib64/R/site-library"            "/opt/R/4.0.3/lib64/R/library"
>
```

Figure 6 Loading bundle outputs version 1.1.1

In the above example, comparing `.libPaths()` before and after, the bundle load highlights the addition of the bundle R package library. Note that in this example, the user library is prior to the bundle in the search sequence which implies that any packages installed in both locations, the version in the user library takes precedence.

```
> .libPaths()
[1] "/home/user/R/x86_64-pc-linux-gnu-library/4.0" "/opt/R/4.0.3/lib64/R/site-library"
[3] "/opt/R/4.0.3/lib64/R/library"
>
> cxlib_bundle_load( "stats/1.2.3" )
>
> .libPaths()
[1] "/opt/bundles/stats/1.2.3/libraries/R/4.0.3"   "/opt/bundles/outputs/1.1.1/libraries/R/4.0.3"
[3] "/opt/R/4.0.3/lib64/R/site-library"            "/opt/R/4.0.3/lib64/R/library"
>
```

Figure 7 Loading bundle stats version 1.2.3

The more complex *stats* bundle (Figure 7) uses the same syntax, but with some different results. The R package library of the *stats* bundle is added to `.libPaths()` and so is the R package library for the *outputs* bundle. The *stats* bundle requires that specific version of the *outputs* bundle as defined in the *stats* 1.2.3 bundle R properties file.

Also note that the user library is no longer included in `.libPaths()`. The user library was disabled through *stats* bundle main properties, which can be used to ensure that only a predefined set of packages, e.g. those within the loaded bundles, site library and library, are available for use.

In the previous examples, we refer to the version of the bundle to load. To load the default version of a bundle, simply refer to the bundle by name only (Figure 8) and the bundle load mechanism will attempt to resolve and load the default version. If no default version is defined, we simply get an error. And, of course, the default bundle version is a property in the bundle main R properties file, permitting defaults to be assigned by language. Note that we also explicitly disable the user library in this example,

```
> .libPaths()
[1] "/home/user/R/x86_64-pc-linux-gnu-library/4.0" "/opt/R/4.0.3/lib64/R/site-library"
[3] "/opt/R/4.0.3/lib64/R/library"
>
> cxlib_bundle_load( "outputs", user.library = FALSE )
>
> .libPaths()
[1] "/opt/bundles/outputs/1.1.1/libraries/R/4.0.3" "/opt/R/4.0.3/lib64/R/site-library"
[3] "/opt/R/4.0.3/lib64/R/library"
>
```

Figure 8 Loading default version of the outputs bundle

The same principles can be applied to other languages, say SAS to load a SAS macro library through amending SASAUTOS, as just one example.

```
%cxlib_bundle_load( bundles = output/1.1.1 );
```

The equivalent mechanism for Python would be to amend `sys.path` or PYTHONPATH depending on where in the execution sequence the Python module is loaded.

CREATING A BUNDLE

The steps to create a bundle is no different than installing packages in an alternative location. As an example, below (Figure 9) represents a simple 4-step approach to create the *outputs* bundle, performed entirely from within R.

```
# create the bundle package library directory
dir.create( "/opt/bundles/outputs/2.3.4/libraries/R/4.0.3", recursive = TRUE )

# create the r.properties file ... can be empty
writeLines( "", con = "/opt/bundles/outputs/2.3.4/r.properties")

# load the bundle to update .libPaths()
cxlib_bundle_load( "outputs/2.3.4", user.library = FALSE )

# install packages
install.packages( ... )
```

Figure 9 Manually create a bundle

The example also explicitly disables the user library when loading the empty *outputs* bundle to ensure that the *outputs* R package library path is the first entry in `.libPaths()` and any packages installed in the user library does not affect the bundle package installation.

As such, creating a bundle can simply rely on the current practices within your organization for distributing and installing R packages, whether that is using `install.packages()` or any other custom deployment process.

CONCLUSION

There are many different approaches to R package management. The flexibility of R can affect the effort further as additional business requirements and constraints may be imposed by an organization's Good Clinical Practice (GCP), or other wider GxP, quality and compliance standards.

A simple use of directory structures and attributes that relies on standard R functionality can be employed as a simple mean to manage multiple R libraries concurrently and at the same time minimize disruption, retain flexibility of smaller curated package collections, and reduce the validation effort.

We can at the same time continue using the same mechanisms, tools, and functions to install and maintain packages, creating an architecture that can be used within existing infrastructure and compliance processes.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Magnus Mengelbier
Limelogic AB
papers at limelogic.com
limelogic.com