Paper OS14

Using Bundles for R Package Management

Magnus Mengelbier, Limelogic, Malmö, Sweden

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 history with R. The task grows in complexity as the number of packages increase and, at some point, we need to update an existing package because the current version is too old, you want to use a new feature, there is a conflict with other packages, or it simply stops working.

Consider the same scenario, not for your personal package libraries, but packages installed in a central library accessible to all other users, such as the R application or site library. Most likely the number of packages is far greater as it needs to accommodate all use cases and any disruption is bound to delay delivering analysis.

An extremely popular approach for most users and organizations is to use user or project driven package managers, such as *renv* or *packrat*, either standalone or together with commercial package management solutions. The choice of package managers and the maintenance effort is further affected by additional business requirements and constraints to comply with an organization's Good Clinical Practice (GCP), the wider GxP, or other quality and compliance standards.

The principle of a bundle is extended to R, giving a simple means to manage multiple libraries concurrently while minimizing disruption, retaining 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 PACKAGE 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 in a solution, 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/or private user library.

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

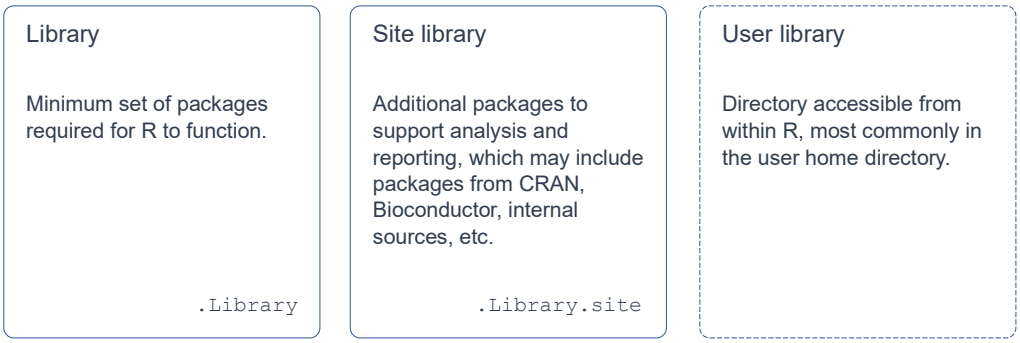

Figure 1 Standard package libraries

The most common configuration for R environments (Figure 1) is to create and maintain a central Site library and permit users to install packages in their User library.

The Library is where R maintains the minimum set of packages required to function. Depending on the validation approach, the minimum set of packages installed by default 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 provide statistical methods as well as being under development and/or maintenance, so updates are periodically released independently of new R versions beyond a highly likely GCP validation requirement.

The preferred approach is for any additional or add-on R packages to be installed in the R application Site library (paths 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 centrally shared compute and analysis capabilities. It is important to note that Site Library is usually referred to in the singular even though that it can be multiple paths.

The User library may also be disabled in GCP compliant R environments as organization compliance and validation standards may require that packages are installed, validated and immutable, i.e. read-only, which would negate the use of the User library.

In practice, there is one additional set of locations where packages can be installed. That is any directory that is accessible from within the executing R session and that it can be included in the library tree, i.e. a member of `.libPaths()`. That the library tree can be updated from within an R session is the principal mechanism that lets a user use a package in a bundle during programming and analysis.

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 end of 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 aspect is that the R application Library and Site library are by default accessible to all users, meaning any package updates in those libraries has a broad impact that needs to be carefully considered and coordinated. Both the complexity and disruption significantly increase as the number of packages grow given packages natural evolving, and sometimes intricate, dependencies.

Package managers like the mentioned *renv* and *packrat* circumvents this by overriding the R configuration with a user session's personal application Library and Site library, which simply means any previously installed packages are hidden from the user and not directly accessible unless manually added to the library tree. It also means that each *renv* or *packrat* project will need to re-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. This discussion would also address how those compliance requirements and principles apply to packages and whatever package managers are used, bundles included. As a broad statement, we assume that an organization has determined that R will need some level of validation to be GCP compliant, and that 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, like R, is expected to meet with respect to GCP, or any other of the GxP domains for that matter. GAMP principles can be combined with the *riskmetric* package or similar schemes to provide a systematic approach to package validation.

R, as a mathematical and/or statistical language, could fall within either GAMP v5 Category 1 or 3, both categories 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 and extended by the R packages you install, so just validating and using R without packages would not provide the analysis capability you need and expect.

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. Languages constructs such as generic functions and methods cannot be disabled so R principles of modularity is here to stay.

Modularity 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. More so, it is also dependent if and in what sequence that a user loads a package into the namespace, i.e. `library()` and `require()` functions. Modularity is thereby defined by the language constructs and how they are 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 in theory 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 large number of permutations of different packages and respective versions.

An organization may require that the modular aspect of the R environment is incorporated in the validation of packages to demonstrate that the install sequence, namespace load sequence and conflicts originating with 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 the native capability of R to 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 on Unix/Linux and classic DLL's on Microsoft Windows environments.

```
* 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 survival package

This extends dependencies beyond packages to also include utilities, such as the GNU c/c++ compiler (gcc above), and shared/dynamic libraries not part of the R distribution and the R package itself. This chain of external dependencies, commonly referred to as a *tool chain*, also needs to be considered as part pre-requisite and dependencies when packages are validated and installed. As those dependencies are most likely updated independently of the R application itself and its packages, say during IT security patching, the tool chain dependencies need to be adequately managed. If the GCP qualified R environment is frequently updated and users are permitted to install packages, the likelihood of external dependencies breaking package functionality increases.

Consider the simple case where *renv* is used to manage the library of validated packages for an analysis. Let us assume that the package was validated at some point in time, IT performs patching and other updates to the R environment that includes an external dependency for the validated package. Is the package installed for this first analysis still validated if it was installed prior to the patch?

Now assume a new analysis project installs the same validated package using *renv*. Is the package still validated given that the external logic it relies on was updated by the IT update? In most organizations, the answer is simply a plain *no* and others may say possibly *yes*. The latter *yes* is likely conditional on if the IT patch did or did not impact to change the functionality of the packages, which could only be demonstrated by ensuring that any of the patched components are not used by the package (an external dependency may have its own

dependencies). The easiest and least resource intensive way to provide the expected evidence would be to simply re-run the validation after IT patching.

If, on the other hand, users and projects are not permitted to install packages or use package managers like *renv and packrat*, then only packages that have been installed and validated prior to use will be available. This would imply that the R environment is based on a pre-defined set of capabilities and classed as non-configurable software (GAMP v5 Category 3), which is a historically common, well-known and acceptable software category within Life Sciences.

As packages most often 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.

The compliance focus can then be on the packages themselves. As the packages are considered part of the application, they would follow the GAMP v5 Category 3 categorization. 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).

This is however a static set of capabilities without the benefits of the modularity of R and the inherent flexibility provided by packages managers like *renv* and *packrat*. As such, we really have not solved the main issue when 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. Experience has demonstrated that the maintenance and validation effort grow unproportionately with an increase in the number of packages.

Separating *related* packages into smaller libraries, i.e. bundles, will allow an organization to minimize complexity, retain a more straightforward efficient validation process for non-configurable software and at the same time retain some of the benefits of the modularity of R.

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.

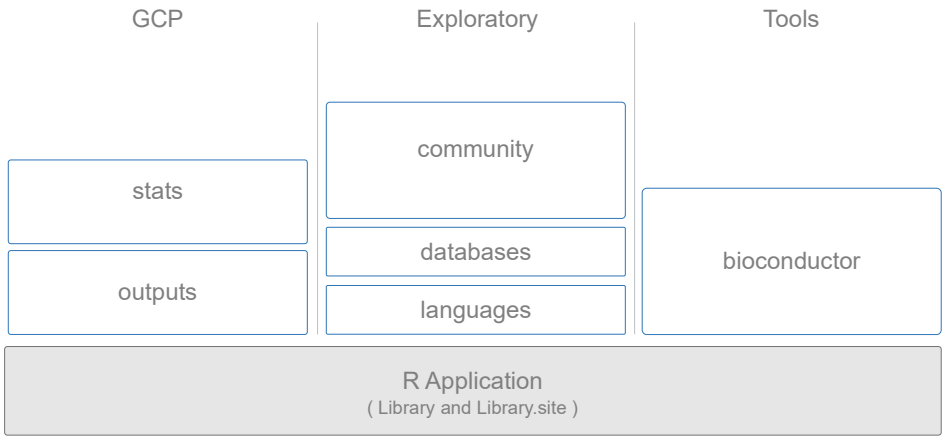

Figure 3 Bundles concept

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 and associated processes. 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, if we can maintain different versions of a bundle, it could possibly limit disruption due to maintenance and significantly improve reproducibility.

Disruption to the business when central library tools and utilities are updated, and not just with R, is a common occurrence and the timing is seldom perfect for all parties involved. Bundles has the potential to eliminate disruptions when packages and their dependencies are updated, if it is permitted that a study or project can

continue to use an older validated version of a bundle. Any bundle updates could then be released as new versions, not impacting what has previously been released.

Reproducibility could also become a simple exercise. Instead of attempting to recreate a Site library in some alternative location, the study or project teams could simply just refer to the pre-existing bundle version, with the understanding that versions of a bundle are only deleted when explicitly required. The process of re-creating the analysis could then be further simplified if at a certain study or project milestone, business processes mandate that the analysis *locks* to a specific version of the bundles used during an analysis and further avoiding any unnecessary edits to analysis programs.

BUNDLE ARCHITECTURE

There are different approaches to discuss the architecture of a bundle with the most straightforward being from the user perspective and then deconstruct the different layers and elements.

```
cxlib_bundle_load( "outputs/1.1.1", user.library = FALSE )

cxlib_bundle_load( "outputs", user.library = TRUE )
```

Figure 4 Example loading a bundle

The first statement above *loads* the bundle `outputs/1.1.1`. An R bundle is inherently by design a dynamically loadable R package library. In addition. the first statement also highlights two important constructs in that a bundle is named, e.g. *outputs*, and a bundle is versioned, i.e. *1.1.1*.

The second statement also loads bundle *outputs* and demonstrates the concept of the default version. This does not necessarily mean the latest version, but a version that is designated as default.

```
> .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 5 .libPaths() and loading a bundle

Both statements load the *outputs* bundle, search for the requested version and adds the bundle's R package library, i.e. …`/outputs/1.1.1/libraries/R/4.0.3`, to the library tree (Figure 5).

The bundle concept relies entirely on the principles and standard functionality for R libraries and does not require any additional packages beyond the minimal default set.

```
cxlib_bundle_load( "outputs/1.1.1", user.library = FALSE )

/opt/bundles/outputs/1.1.1/libraries/R/4.0.3
Install directory          R package library
```

Figure 6 Bundle path structure

Breaking down the bundle's R package library path (Figure 6) added to the library tree (Figure 5) highlights the underlying mechanisms at work. Bundles are installed in one or more directories that is part of the bundle search paths. Similar to the search for packages, the first occurrence that matches the bundle request is used, meaning the *outputs* bundle may be installed in any of the directories in the search path, a technique that can be used for development, pilots, testing, validation, production and so forth.

Any subdirectory in any of the bundle search paths could be a potential bundle. An R bundle is therefore identified through the existence of the `libraries/R` directory structure, or the lower-case equivalent. This also enables a bundle to be installed in the same location where it could support languages other than R, such as SAS or Python. A bundle is thereby by design multi-lingual.

The last directory in the path denotes a specific version of R. This allows the same version of a bundle to support multiple versions of R as a particular R version may not be able to read or use packages installed by a previous

older version, the R versions may be on different tool chains or simply the result of compliance requirements that packages, and thus the bundle, are validated for a specific pre-determined version of R.

The above discussion has however assumed that the version of the bundle is specified. If you recall the second statement in our initial load example, bundles support the notion of a default version. Technically, the default version can be resolved in multiple ways. The most common implementation has been creating the symbolic link *default* in the bundle name directory pointing to the designated default version. Traditionally, Microsoft Windows has not supported the concept of a symbolic links[1], so a simple text or properties file in the bundle name directory can be used just as well.

BUNDLE PROPERTIES

The basic principles of a bundle can be further extended by associating properties. As an example, we consider the *cxlib* package and its mechanism of choice. It uses a standard properties file, consisting of named key and value pairs. This format is directly usable within R using the default set of packages, i.e. function `read.table()` in the *utils* package. This further supports the principle that bundles should not require any additional packages beyond the R default set.

With the *cxlib* package, bundle properties can be defined on multiple levels. Properties can be defined for each R version, across all R versions, for the bundle version and for all R bundles and with that order of precedence.

A useful example is bundle dependencies, where one bundle depends on and automatically loads another bundle (Figure 7). Dependencies on other bundles or an external resource is commonly defined for each bundle version due to validation constraints. In our example, loading the `stats/1.2.3` bundle automatically loads the `outputs/1.1.1` bundle as a dependency. Note that the User library is disabled, controlled though bundle properties of either *stats*, *outputs* or both.

```
> .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 Example bundle dependency with additional controls

Properties can also be used to pre-define a scope of use, akin to a license, say a bundle may be permitted for use in exploratory analysis but not for GCP analysis and deliverables. This simple idea could allow GCP environments to also permit the less restrictive, and potentially unvalidated, use of packages for exploratory analysis while still retaining the stricter compliance criteria for GCP workloads. In *cxlib*, and other similar implementations, the loading mechanism contains a configurable restriction to ensure that GCP and exploratory licensed bundles cannot be loaded at the same time, and/or that it is clearly noted in the log that a bundle for exploratory use only is loaded.

Other properties, both general and version specific, could include properties for supported versions of R, workflow states, search paths, business process specific configurations, etc.

CREATING A BUNDLE

The steps to create a bundle is no different than installing packages in an alternative location. As an example, below represents a simple R-native approach to manually create the *outputs* bundle using `install.packages()`.

[1] See Microsoft Windows command mklink

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

# -- note: create any property files here if your bundle has dependencies

# load the emmpty bundle
# note: user.library is disabled so that any user library packages are ignored
cxlib_bundle_load( "outputs/2.3.4", user.library = FALSE )

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

Figure 8 Manually creating a bundle

The bundle directory is created followed by loading the bundle with the User library disabled. As the User library is by convention the first path in the library tree, this ensure us that the bundle path is now first and the default install location used by `install.packages()`.

The above approach works well with a single environment or when the install program is tightly scripted to ensure that you meet the desired consistency across environments, but it does have its logistical limitations. As the number of packages increase and you have to accommodate for the nuances of every environment, the script become difficult to manage and use.

```
# create initial bundle reference
# note: ... or read in an existing specification
mybundle <- cxlib_bundle( name = "outputs")
mybundle$version( "2.3.4" )

# disable user library
mybundle$option( "r/user.library" = FALSE )

# add packages
mybundle$add.packages( ... )

# create the speficiation file
mybundle$save()
```

Figure 10 A bundle specification

A better approach would be to construct a bundle specification (Figure 9) and then deploy the specification to each target environment (Figure 10). The specification can be a simple text file, such as JavaScript Object Notation (JSON) format used by *cxlib*. The specification only needs to define a limited minimal set of properties, such as the names of each included package and its respective version.

The result of using specifications in deployment is a simplified process that ensures that each version of a bundle is identical across every environment that has it deployed, most likely down to the version of each individual

```
# deploy outputs
# note: assuming that the deploy sources have been defined in options
mydeploy <- cxlib_bundle_deploy( "outputs/2.3.4" )
```

Figure 9 Deploying a bundle specification

package.

It is also directly feasible that the bundle specification can be shared with third parties, community collaborators and regulatory authorities. In most implementations, the specification and install sources are shared via a controlled GxP compliant archive, such as a GitHub, an Artifactory, a mini CRAN or some comparable repository.

In reality, it may not be possible to use a repository hosted and maintained by your or a third-party organization. As an example, the `cxlib_deploy()` function works around this using a local file-based repository as the source for deployments. Transfers of the specifications and install sources can then be performed by any permitted and appropriate means, even base64 encoding the install sources so they can be transferred as regular text files.

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.

Defining a bundle as a specification and using standard deployment fools and utilities can provide a simple approach to standardize R environments within an organization and, at the same time, be a simple way to share what packages are required for a project or an analysis, but still adheres to compliance requirements.

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 multilingual infrastructure and compliance processes.

ACKNOWLEDGEMENTS

The bundle license concept was devised and proposed by Per Arne Stahl at AstraZeneca.

CONTACT INFORMATION

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

Magnus Mengelbier
magnus.mengelbier@limelogic.com

Limelogic AB
Jungmansgatan 12
211 11 Malmö
Sweden

www.limelogic.com
github.com/limelogic