Writing Frictionless R Package Wrappers

Bob Rudis

2020-01-02

Contents

Chapter 1

Preface

Chapter 2

Introduction

The R language and RStudio IDE are a powerful combination for "getting stuff done", and one aspect of R itself that makes it especially useful is the ability to use it with other programming languages via a robust *foreign language interface* capability[1]. The term "foreign language" refers to another programming language such as C, C++, Fortran, Java, Python, Rust, etc. A common way of referring the this idiom of using functionality written in another programming language from directly within R is "wrapping" since we're putting an R "shell" around the code from the other language. Another term you may see used is "extending" (hence the title of the "Writing R Extensions" R manual).

While R supports using this this extension mechanism from any R script leaving tiny trails of R and other language source and binary files all across your filesystem is not exactly the best way to keep these components organized and creates other challenges when you come across the need to use them in other projects or share them with others. Thankfully, the R Core team, along with many individual contributors over the years, has made it pretty straightforward to incorporate this extension capability into R packages which are much easier (honest!) to organize and share.

The goal of this book is to help you get up to speed using R and RStudio to write R packages that wrap code from many different languages to help you "get stuff done" with as little friction as possible.

[1]"Writing R Extensions"; Chapter 5, "System and foreign language interfaces"; (https://cran.r-project.org/doc/manuals/r-release/R-exts.html#System-and-foreign-language-interfaces)

2.1 Base Requirements

It is assumed that readers are familiar with the R programming language, RStudio IDE, and are comfortable installing and using packages. Since this work is about extending R with other programming languages, you should also have some knowledge of one or more of the target languages being covered.

To follow along with the series you'll need to ensure you have the necessary components installed along the way. Rather than overwhelm you with all of them up front, each new section will introduce requirements specific to the language or situation being covered. However, there are some fundamentals you'll need to ensure are available.

- An R[2] environment, preferably R 3.6.x which is what was used for this series.
- RStudio[3], as we'll be using many of the features provided in it to help reduce development friction
- The {pkgbuild}[4] package installed

Once you've gotten through those steps, you should fire up RStudio and run:

```r
pkgbuild::check_build_tools(debug = TRUE)
```

which will help you make sure your particular system is ready to build packages.

After performing the build tool check and/or installation of the necessary core tools, you will then need to install the {devtools}[5] package, which will help ensure that the remaining core packages required are installed.

We're also going to use the `git`[6] source code version control system. The `git` ecosystem is *not* "GitHub", which is just a public (or, potentially somewhat private) place to house source code repositories, just like other hosted services such as Bitbucket, GitLab, or SourceHut. You can use the excellent "Happy Git with R"[7] resource to help ensure you're source code control environment is also ready to use.

2.2 Supplemental References

It may be helpful to create a browser bookmark folder for supplemental reference material that will be referred to from time-to-time across the sections (we'll be adding to this list in each chapter, too):

[2] R Project Home (https://www.r-project.org/)
[3] RStudio Home (http://rstudio.com/)
[4] {pkgbuild} CRAN page (https://cran.rstudio.com/web/packages/pkgbuild/)
[5] {devtools} Home (https://devtools.r-lib.org/)
[6] `git` Home (https://git-scm.com/)
[7] Happy Git with R (https://happygitwithr.com/)

- Writing R Extensions (https://cran.r-project.org/doc/manuals/r-release/R-exts.html)
- Advanced R (http://adv-r.had.co.nz/)
- R Packaged (http://r-pkgs.had.co.nz/)

2.3 Up Next

If you've been a user of "development versions" of R packages or have authored R packages you likely made quick work of this first installment. Those new to creating packages with R, those who tend to only use fully-baked CRAN versions of R packages, and/or those who have not worked with `git` before likely had to do quite a bit of work to get down to this point (if this describes you, you definitely deserve both a break and kudos for getting this far!).

In the next installment we'll make sure the package building infrastructure is ready to roll by creating a basic R package that we'll use as a building block for future work.

Chapter 3

Building A Basic R Package

Before we start wrapping foreign language code we need to make sure that basic R packages can be created. If you've followed along from the previous chapter you have everything you need to get started here. *Just to make sure*, you should be able to fire up a new RStudio session and execute the following R code and see similar output. If not, you'll need to go through the steps and resources outlined there before continuing.

```
pkgbuild::check_build_tools()
## Your system is ready to build packages!
```

3.1 Configuring {devtools}

We're going to rely on the {devtools} package for many operations and the first thing you should do now is execute `help("create", "devtools")` in an RStudio R console to see the package documentation page where you'll see guidance pointing you to `devtools::use_description()` that lists some R session `options()` that you can set to make your package development life much easier and quicker. Specifically, it lets you know that you can setup your `~/.Rprofile` to include the certain options settings which will automatically fill in fields each time you create a new package vs you either specifying these fields manually in the package creation GUI or as parameters to `devtools::create()`.

A good, minimal setup would be something like:

```
options(
  usethis.description = list(
    `Authors@R` = 'person("Some", "One", email = "someone@example.com", role = c("aut", "cre"),
                          comment = c(ORCID = "YOUR-ORCID-ID"))',
    License = "MIT + file LICENSE"
  )
)
```

NOTE: If you do not have an "ORCID" you really should get one (they're free!) by heading over to — https://orcid.org/ — and filling in some basic information.

Take a moment to edit your `~/.Rprofile`. If you're not sure about how to do that there is an excellent chapter in Efficient R Programming[1] which walks you through the process.

Once you've added or verified these new `options()` settings, restart your R session.

3.2 Creating A Package

We're *almost* ready to create and build a basic R package. All R packages live in a package directory and I highly suggest creating a `packages` directory right off your home directory (e.g. "`~/packages`") or someplace where you'll be able to keep them all organized and accessible. The rest of these chapters will assume you're using "`~/packages`" as the

With {devtools} now pre-configured, use the RStudio R Console pane to execute the following code which should produce similar output and open up a new RStudio session with the new package directory:

```
devtools::create("~/packages/myfirstpackage")
##   Creating '/Users/someuser/packages/myfirstpackage/'
##   Setting active project to '/Users/someuser/packages/myfirstpackage'
##   Creating 'R/'
##   Writing 'DESCRIPTION'
## Package: myfirstpackage
## Title: What the Package Does (One Line, Title Case)
## Version: 0.0.0.9000
## Authors@R (parsed):
##     * Bob Rudis <bob@rud.is> [aut, cre] (<https://orcid.org/0000-0001-5670-2640>)
## Description: What the package does (one paragraph).
## License: MIT + file LICENSE
## Encoding: UTF-8
## LazyData: true
##   Writing 'NAMESPACE'
##   Writing 'myfirstpackage.Rproj'
##   Adding '.Rproj.user' to '.gitignore'
##   Adding '^myfirstpackage\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
```

[1] Efficient R Programming, "3.3 R Startup"; (https://csgillespie.github.io/efficientR/3-3-r-startup.html#r-startup)

```
##   Opening '/Users/someuser/packages/myfirstpackage/' in new RStudio session
##   Setting active project to '<no active project>'
```

The directory structure will look like this:

```
.
  DESCRIPTION
  NAMESPACE
  R/
  myfirstpackage.Rproj
```

At this point we still do not have a "perfect" R package. To prove this, use the R console to run `devtools::check()` and — after some rather verbose output — you'll see the following lines at the end:

```
> checking DESCRIPTION meta-information ... WARNING
  Invalid license file pointers: LICENSE

0 errors   | 1 warning x | 0 notes
```

Since we're saying that our package will be using the MIT license, we need to ensure there's an associated `LICENSE` file which we can do by executing `usethis::use_mit_license()` which will create the necessary files and ensure the `License` field in the `DESCRIPTION` file is formatted properly.

If you run `devtools::check()` again, now, your final line should report:

```
## 0 errors   | 0 warnings   | 0 notes
```

and the package directory tree should look like this:

```
  DESCRIPTION
  LICENSE
  LICENSE.md
  NAMESPACE
  R/
  myfirstpackage.Rproj
```

3.3 Rounding Out The Corners

While we have a minimum viable package there are a few other steps we should take during this setup phase. First we'll setup our package to use `{roxygen2}`[2] for documenting functions, declaring `NAMESPACE` imports, and other helper-features that will be introduced in later chapters. We can do this via `usethis::use_roxygen_md()`:

[2]{roxygen2} Home; (https://roxygen2.r-lib.org/)

```
usethis::use_roxygen_md()
##   Setting Roxygen field in DESCRIPTION to 'list(markdown = TRUE)'
##   Setting RoxygenNote field in DESCRIPTION to '7.0.2'
##   Run `devtools::document()`
```

We won't run `devtools::document()` *just yet*, though. Before we do that we'll also create an R file where we can store top-level package introduction/meta-information:

```
usethis::use_package_doc()
##   Writing 'R/myfirstpackage-package.R'
```

Now, our directory tree should look like:

```
.
  DESCRIPTION
  LICENSE
  LICENSE.md
  NAMESPACE
  R
     myfirstpackage-package.R
  myfirstpackage.Rproj
```

Now, run `devtools::document()` which will translate the {roxygen2} comments into a properly-formatted R documentation file and regenerate the `NAMESPACE` file (as we'll be managing package imports and exports via {roxygen2} comments). The directory tree will now look like:

```
.
  DESCRIPTION
  LICENSE
  LICENSE.md
  NAMESPACE
  R
     myfirstpackage-package.R
  man
     myfirstpackage-package.Rd
  myfirstpackage.Rproj
```

and, we can now re-run `devtools::check()` to make sure we have the three "0's" we're aiming for each time we check our package for errors.

3.4 Passing The Test

We're going to want to write and use tests to ensure our package works properly. There are many R package testing frameworks available. To ease the introduction into this process, we'll use one of the frameworks that came along for the ride when you installed the various packages outlined in the previous chapter: {testthat}[3]. Setting up {testthat} is also pretty painless thanks to the {usethis} package we've been taking advantage of quite a bit so far. We'll create the {testthat} overall infrastructure then add a placeholder test script since `devtools::check()` will complain about no tests being available if we do not have at least a single script it can execute during the test phase of the package checking process.

```
usethis::use_testthat()
##   Adding 'testthat' to Suggests field in DESCRIPTION
##   Creating 'tests/testthat/'
##   Writing 'tests/testthat.R'
##   Call `use_test()` to initialize a basic test file and open it for editing.

usethis::use_test("placeholder")
##   Increasing 'testthat' version to '>= 2.1.0' in DESCRIPTION
##   Writing 'tests/testthat/test-placeholder.R'
##   Modify 'tests/testthat/test-placeholder.R'
```

The directory tree will now look like this:

```
.
  DESCRIPTION
  LICENSE
  LICENSE.md
  NAMESPACE
  R
     myfirstpackage-package.R
  man
     myfirstpackage-package.Rd
  myfirstpackage.Rproj
  tests
      testthat
         test-placeholder.R
      testthat.R
```

Run `devtools::check()` one more time to make sure we've got those precious 3 "0's" one last time.

3.5 Getting Things Under Control

We're *almost* done! One final step is to turn this directory into a git-managed directory so we can work a bit more safely and eventually share our development work with a broader audience. Provided you followed the outline in the previous chapter, setting up git is as straightforward as one {usethis} function call:

[3]{testthat} Home; (https://testthat.r-lib.org/)

```
usethis::use_git()
##   Setting active project to '/Users/someuser/packages/myfirstpackage'
##   Initialising Git repo
##   Adding '.Rhistory', '.RData' to '.gitignore'
## There are 10 uncommitted files:
## * '.gitignore'
## * '.Rbuildignore'
## * 'DESCRIPTION'
## * 'LICENSE'
## * 'LICENSE.md'
## * 'man/'
## * 'myfirstpackage.Rproj'
## * 'NAMESPACE'
## * 'R/'
## * 'tests/'
## Is it ok to commit them?
##
## 1: For sure
## 2: Negative
## 3: Not now
##
## Selection: 1
##   Adding files
##   Commit with message 'Initial commit'
##   A restart of RStudio is required to activate the Git pane
## Restart now?
##
## 1: Negative
## 2: Not now
## 3: Yup
##
## Selection: 3
```

RStudio should have been restarted (so it can add a "Git" pane in case you want to use the GUI to manage git) and the directory tree will now have a `.git/` subdirectory that you should (almost) never touch by hand.

The last thing to do is to "vaccinate" your git setup so you don't leak sensitive or unnecessary files when you (eventually) share your creation with the world:

```
usethis::git_vaccinate()
##   Adding '.Rproj.user', '.Rhistory', '.Rdata' to '/Users/someuser/.gitignore'
```

We now have a basic, working R package that is devoid of any real functionality other than that of getting us familiar with the package setup and validation processes. We'll be building upon this experience in most of the coming chapters.

3.6 Quick Reference

After ensuring you've got the recommended `options()` in place, here are the steps to setup a new package:

```r
# in any RStudio R Console session
devtools::create("~/packages/THE-PACKAGE-NAME")

# in the newly created package RStudio R Console session:
usethis::use_mit_license()       # need a LICENSE file
usethis::use_roxygen_md()        # use {roxygen2} for documentation and configuration
usethis::use_package_doc()       # setup a package-level manual page
usethis::use_testthat()          # setup testing infrastructure
usethis::use_test("placeholder") # setup a placeholder test file
devtools::document()             # Let {roxygen2} create NAMESPACE entries, build manual pages (
devtools::check()                # looking for the three "0's" that tell us we're ready to roll!
usethis::use_git()               # put the directory under git version control
usethis::git_vaccinate()         # Prevent leaking credentials and other unnecessary filesystem
```

Rather than re-type `devtools::document()` (et al) whenever you need to run {roxygen2} or build/check a package you can use RStudio keyboard shortcuts that are designed to seamlessly integrate with the {devtools} ecosystem:

Operation	Windows & Linux	Mac	{devtools} equivalent
Install and Restart	Ctrl+Shift+B	Cmd+Shift+B	devtools::install()
Load All (devtools)	Ctrl+Shift+L	Cmd+Shift+L	devtools::load_all()
Test Package (Desktop)	Ctrl+Shift+T	Cmd+Shift+T	devtools::test()
Test Package (Web)	Ctrl+Alt+F7	Cmd+Alt+F7	devtools::test()
Check Package	Ctrl+Shift+E	Cmd+Shift+E	devtools::check()
Document Package	Ctrl+Shift+D	Cmd+Shift+D	devtools::document()

We'll refer to these operations as "install" (or "build"), "load all", "test", "check", and "document" from now on so you can choose to use the console or the shortcuts as you prefer.

3.7 Exercises

Our package may be kinda, well, *useless* for the moment but that doesn't mean you can't show it some love and get some practice in at the same time while things are still relatively straightforward.

- Modify the `Title`, `Version`, and `Description` fields of the `DESCRIPTION` file and refine them as needed until package checks pass.

- Deliberately mangle parts of the `DESCRIPTION` file to see what errors or warnings you receive during the package check process.
- Read up on {roxygen2} and add some `Section`s to it formatted with markdown and/or LaTeX. Re-“document” the package and see how your changes look.
- Edit the `test-placeholder.R` file and change the placeholder test it created so it fails and then re-check the package to see what warnings or errors show up.
- After you’ve made (valid, working) modifications to any/all of the above *and package checks pass*, use either the git command line tools or the RStudio Git pane to add your updates to the git tree. Use the resources linked to in the previous chapter if you need a refresher on how to do that.
- Re-run through all the steps with a brand new package name just to make sure you’re comfortable with the package creation process.

3.8 Up Next

In the next installment in the series we will start wrapping by creating a basic wrapper that just calls out to the operating system shell to run commands.