Managing R Packages with roxyPackage

m.eik michalke

April 7, 2013

R packages consist of numerous files in various formats, like `*.R`, `*.Rd`, `DESCRIPTION`, `NAMESPACE` or `CITATION`, ordered in a specific directory structure. `roxyPackage` tries to automatically generate most of these structures and files, so that you can concentrate on writing the actual R code for your package. If your code includes `roxygen2`-style documentation, this package can update your docs as well. It is also capable of creating a local package repository, including limited support for generating Mac OS X, Windows and Debian packages.

1 Getting started

Apart from installing `roxyPackage` and having written some R code, no further requirements for building your first R package need to be fulfilled. I tried to keep the dependencies for `roxyPackage` to a minimum, and basically they come down to `roxygen2`[1] (used to generate the documentation) and `XiMpLe`[2] (used to generate the HTML and RSS files of the repository).

To demonstrate this, we will start with an extremely simple one: Let's assume you would like to have an R package called "examplePackage" and have just written a first function for it, called "simpleExample", which does nothing but return `NULL`:

```
simpleExample <- function(){NULL}
```

To transform this code into a package, save it as a file called `"examplePackage/R/simpleExample.R"`, that is, create a directory with the package name and in it another one named `"R"`. If you were to create the package manually,[3] you would now at least write a `DESCRIPTION` file with meta information like the license, version number and release date, a `NAMESPSACE` file to export the function so it can acually be

[1] `http://cran.r-project.org/web/packages/roxygen2`
[2] `http://www.reaktanz.de/?c=hacking&s=XiMpLe`
[3] see `http://cran.r-project.org/doc/manuals/R-exts.html`

seen and used after the package was loaded, and a `*-package.Rd` file, which has the meta information in Rd format so it's accessible via the R help system. In a minute, all of this will be done automatically by `roxyPackage`.

Before this, we need to add at least one `roxygen2` command to the function code, which will make sure that the function is exported and user accessible:

```
#' @export
simpleExample <- function(){NULL}
```

This must be done for every function, class etc. you would like to export. But that's about it – we can now call `roxy.package()` to create a valid R package. Of course you have to tell the function some things:

- where is the source directory of the package (`"~/myRcode/examplePackage"`)
- where is the R library the package can be installed to (e. g., `"~/R"`)
- in which directory should the repository be created (e. g., `"~/myRcode/repo"`)
- what is the package version number, description, author, its dependencies etc.

This is what such a call would look like:

```
roxy.package(
  pck.source.dir="~/myRcode/examplePackage",
  pck.version="0.01-1",
  R.libs="~/R",
  repo.root="~/myRcode/repo",
  pck.description=data.frame(
    Package="examplePackage",
    Type="Package",
    Title="An R Example Package",
    Author="Ernst A. Dölle <e.a.doelle@example.com>",
    AuthorsR="c(person(given=\"Ernst\", family=\"Dölle\",
      email=\"e.a.doelle@example.com\",
      role=c(\"aut\", \"cre\")))",
    Maintainer="Ernst A. Dölle <e.a.doelle@example.com>",
    Depends="R (>= 2.10.0)",
    Description="Provides a great function to produce NULL results.",
    License="GPL (>= 3)",
    Encoding="UTF-8",
    LazyLoad="yes",
    URL="http://example.com/doelleInnovations",
    stringsAsFactors=FALSE))
```

The `data.frame` which is used for the `pck.description` can include every valid field of R `DESCRIPTION` files,[4] except "Version:" and "Date:", as these have their own param-

[4] see `http://cran.r-project.org/doc/manuals/R-exts.html#The-DESCRIPTION-file`

eters (`pck.version` and `pck.date`, which defaults to `Sys.Date()`), and "Authors@R" was simplified to "AuthorsR".

At a first glance this maybe look like a lot. But as you can see, except for of course the version number and release date, most of this information will probably remain the same over time, so you can write all of this in a script and then run the script to generate/update your package.

If you run the above call to `roxy.package()`, the R console will display verbose status information:

```
R environment
  R.home: /usr/lib/R
  R.libs: ~/R
repo: created ~/myRcode/repo/pckg/examplePackage.
Updating collate directive in /home/user/myRcode/examplePackage/DESCRIPTION
Updating namespace directives
Writing examplePackage-package.Rd
repo: created ~/myRcode/repo/src/contrib.

repo: examplePackage_0.01-1.tar.gz copied to src/contrib.
* installing *source* package ‘examplePackage’ ...
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
** testing if installed package can be loaded

* DONE (examplePackage)
build: package built and installed.
Processing packages:
 /home/user/myRcode/repo/src/contrib/examplePackage_0.01-1.tar.gz
done
repo: src/contrib/PACKAGES (unix) updated.
```

Now, if you inspect the project directory, you will see that the new files `DESCRIPTION`, `NAMESPACE`, `R/examplePackage-package.R` and `man/examplePackage-package.Rd` have been added, and you will also find a directory structure in the specified repository location.

2 Why the local repository?

I assumed, if you build a package of your R code you probably want to share your work with others. Of course you can upload your package to CRAN. But there might be reasons to have your own repository, for example, if your release pace is considerably higher than the volunteers running CRAN are able to cope with.

Since `roxyPackage` needed to save its packages somewhere anyway, it seemed obvious to organize this place like an R package repository. So effectively, you can simply copy the generated repository directory to your webserver, and you're done. Packages in that repository can then easily be installed from an R session:

```
> install.packages("examplePackage", repos="http://myRepoURL.example.com")
```

If you're running a Debian GNU/Linux based system, `roxyPackage` can also "debianize"[5] your package, build Debian source and binary packages, and add an additional Debian package repository hosting your R packages (see `"deb"` action below).

3 More actions

The tasks performed at each call of `roxy.package()` are called "actions". The default is to run the actions `"roxy"` and `"package"`, i. e., roxygenize the code documentation and build/install the source package.

There are a number of additional actions, and most of them run out of the box without the need to provide more information:

`"cite"`: Use the data from `pck.description` to write a proper CITATION file.

`"license"`: Try to detect the software license of this package from the "License" info in `pck.description`, and add a copy as the LICENSE file; will let you know if it doesn't know the license you specified.

`"doc"`: Update pdf documentation, including vignettes if found in `inst/doc`.

`"log"`: Generate initial `ChangeLog` or update a present `ChangeLog` file.

`"cl2news"`: Look for a `ChangeLog` file in the package root directory, and translate it into an `inst/NEWS.Rd` file.

`"news2rss"`: Look for an `inst/NEWS.Rd` file and translate it into an RSS feed for the repository; you need to provide a value for the `URL` parameter for this to work, where the URL should point to the root location where your repository is available.

`"html"`: Generate updated HTML index files[6] throughout the repository, very similar to the HTML pages CRAN provides for its packages; will also include links to the RSS feed and provide HTML versions of the citation info and news/changelog.

`"win"`: Build a Windows binary package (only works for pure R packages).

`"macosx"`: Build a Mac OS X binary package (only works for pure R packages).

[5] in accordance with the "Debian R Policy", `http://lists.debian.org/debian-devel/2003/12/msg02332.html`

[6] for a practical example, see `http://R.reaktanz.de/pckg/index.html`

`"deb"`: Build Debian source and/or binary packages (only works on Debian based operating systems); this will also set up a Debian package repository inside the defined repository directory and generate the documentation[7] on how to use it, and also needs the `URL` parameter to work, as well as probably additional configuration via the `deb.options` parameter.

`"check"`: Run `R CMD check` on the package.

So, let's re-run `roxy.package()`, to get a copy of the GPL into our sources, HTML index files for the repository and information on how to cite the package properly:

```
roxy.package(
  pck.source.dir="~/myRcode/examplePackage",
  pck.version="0.01-1",
  R.libs="~/R",
  repo.root="~/myRcode/repo",
  pck.description=data.frame(
    Package="examplePackage",
    Type="Package",
    Title="An R Example Package",
    Author="Ernst A. Dölle <e.a.doelle@example.com>",
    AuthorsR="c(person(given=\"Ernst\", family=\"Dölle\",
      email=\"e.a.doelle@example.com\",
      role=c(\"aut\", \"cre\")))",
    Maintainer="Ernst A. Dölle <e.a.doelle@example.com>",
    Depends="R (>= 2.10.0)",
    Description="Provides a great function to produce NULL results.",
    License="GPL (>= 3)",
    Encoding="UTF-8",
    LazyLoad="yes",
    URL="http://example.com/doelleInnovations",
    stringsAsFactors=FALSE),
  actions=c(
    "roxy",
    "cite",
    "html",
    "license",
    "package"
  ))
```

This is what we're told:

```
R environment
  R.home: /usr/lib/R
```

[7] for a practical example, see `http://R.reaktanz.de/pckg/roxyPackage/deb_repo.html`

```
  R.libs: ~/R
license: saved a copy of the GNU General Public License (GPL) as LICENSE
Updating collate directive in /home/user/myRcode/examplePackage/DESCRIPTION
cite: updated ~/myRcode/examplePackage/inst/CITATION.
cite: updated ~/myRcode/repo/pckg/examplePackage/citation.html.

repo: examplePackage_0.01-1.tar.gz copied to src/contrib.
* installing *source* package ‘examplePackage’ ...
** R
** inst
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
** testing if installed package can be loaded

* DONE (examplePackage)
build: package built and installed.
Processing packages:
 /home/user/myRcode/repo/src/contrib/examplePackage_0.01-1.tar.gz
done
repo: src/contrib/PACKAGES (unix) updated.
html: created CSS file ~/myRcode/repo/pckg/web.css
html: copied RSS image to ~/myRcode/repo/pckg/feed-icon-14x14.png
html: updated ~/myRcode/repo/pckg/examplePackage/index.html
html: updated pckg index ~/myRcode/repo/pckg/index.html
html: updated global index ~/myRcode/repo/index.html
```

3.1 Handling ChangeLogs

`roxyPackage` will try its best to be of help when it comes to handling the `ChangeLog` of your package. The `"log"` action can generate an initial log file in a valid format that the other actions can parse (see 3.1.1). It can also update log entries and add new ones (see 3.1.2).

3.1.1 ChangeLog format

It's a good idea to document the changes you make to your packages from one version to another, i. e., keeping a `ChangeLog`. R packages can have this information in Rd markup format as well, usually in `NEWS.Rd`. `roxyPackage` can translate `ChangeLog`s to `NEWS.Rd`s for you, if the structure of your changelog is understood. A working format is to start each new entry with "Changes in version <version number>", optionally followed by a date string in the format "(YYYY-MM-DD)". After that, each log item needs to be indented with spaces and itemized, e. g., by a dash or asterisk. You can categorize log items by writing the category name with a colon at the beginning of a

line. This is what the inital changelog of our example package could look like:

```
ChangeLog for package examplePackage

Changes in version 0.01-1 (2012-04-22)
added:
  - LICENSE and CITATION files
changed:
  - initial release
```

You might also look at the function `initChangeLog()` to get started with a valid file.

3.1.2 Maintaining a ChangeLog

It's OK to maintain your ChangeLog manually with an editor of your choice, as long as you stick to a valid format. To make this even easier, `roxyPackage` has some dedicated functions and methods to deal with ChangeLogs [8]. Typically, you want to add entries to the package version you are currently working on, to document what has been changed, added or fixed. You can do this directly within a `roxy.package()` call, using the `"log"` action in combination with a list of items given to the `ChangeLog` argument:

```
roxy.package(
  pck.source.dir="~/myRcode/examplePackage",
  pck.version="0.01-1",
  R.libs="~/R",
  repo.root="~/myRcode/repo",
  pck.description=data.frame(
    Package="examplePackage",
    Type="Package",
    Title="An R Example Package",
    Author="Ernst A. Dölle <e.a.doelle@example.com>",
    AuthorsR="c(person(given=\"Ernst\", family=\"Dölle\",
      email=\"e.a.doelle@example.com\",
      role=c(\"aut\", \"cre\")))",
    Maintainer="Ernst A. Dölle <e.a.doelle@example.com>",
    Depends="R (>= 2.10.0)",
    Description="Provides a great function to produce NULL results.",
    License="GPL (>= 3)",
    Encoding="UTF-8",
    LazyLoad="yes",
    URL="http://example.com/doelleInnovations",
    stringsAsFactors=FALSE),
```

[8] have a look at the documentation for `readChangeLog()`

```
  actions=c(
    "roxy",
    "cite",
    "html",
    "license",
    "log",
    "package"
  ),
  ChangeLog=list(
    added=c("new extra NULL feature", "new oblivion matrix"),
    fixed=c("finally resolved the reandom results bug")
  )
)
```

This would generate a new log entry for version 0.01-1 if none is present, otherwise the entries in the named `ChangeLog` list would be appended to the existing entry and its date stamp updated. In the list, the element names are treated as the names of log sections, and each character string as a log entry item of that section. Duplicate items will be ignored automatically, so you can simply append new ones to the list as you see fit. However, it wouldn't hurt to check the log from time to time, to convince yourself it's still in shape.

3.1.3 Generating RSS feeds

RSS technology allows you to spread the news on package updates instantly, after interested users subscribed to the news feed. All you need to define additionally is the `URL` parameter with the web address of your repository. After that, you can combine the actions `"cl2news"`, `"news2rss"` and `"html"` to generate an up-to-date RSS feed[9] from your `ChangeLog` file.

3.2 Archiving packages

At some point, you might want to move old packages to an archive location. This can also be done automatically with this package, by calling `archive.packages()`. If you do that and only include your local repository location as an argument, this function will do nothing but print what it would actually do if you want it to. You can then tweak its parameters to your liking, and if you would like to go for it, set `reallyDoIt=TRUE`.

[9] for a practical example, see `http://R.reaktanz.de/pckg/roxyPackage/RSS.xml`