HOWTO generate repository HTML

S. Falcon

August 28, 2018

1 Overview

This document assumes you have a collection of R packages on local disk that you would like to prepare for publishing to the web. The end result we are going for is:

1. Packages organized per CRAN-style repository standard
2. PACKAGES files created for install.packages access
3. VIEWS file created for generating biocViews
4. A vignette directory created containing the extracted vignette pdf files from each source package in the repository.
5. An html directory created containing html descriptions of each package with links for downloading available artifacts.
6. A simple alphabetical listing index.html file

2 CRAN-style Layout

Establish a top-level directory for the repository, we will refer to this directory as reposRoot. Place your packages as follows:

src/contrib Contains all source packages (*.tar.gz).

bin/windows/contrib/x.y Contains all win.binary packages (*.zip). Where x.y is the major.minor version number of R.

bin/macosx/el-capitan/contrib/x.y Contains the mac.binary.el-capitan (El Capitan) (*.tgz) packages.

You will need the following parameters:

```
> reposRoot <- "path/to/reposRoot"
> ## The names are essential
> contribPaths <- c(source="src/contrib",
+                   win.binary="bin/windows/contrib/3.4",
+                   `mac.binary.el-capitan`="bin/macosx/el-capitan/contrib/3.4")
```

3 Extracting vignettes

The `extractVignettes` function extracts pdf files from inst/doc. The default is to extract to a reposRoot/vignettes.

```
> extractVignettes(reposRoot, contribPaths["source"])
```

4 Generating the control files

The `genReposControlFiles` function will generate the PACKAGES files for each contrib path and also create a VIEWS file with complete info for later use by biocViews.

```
> genReposControlFiles(reposRoot, contribPaths)
```

5 Generating the HTML

The `writeRepositoryHtml` will generate HTML detail files for each package in reposRoot/html. The function will also create an index.html file at the top level.

Two CSS files are included with *biocViews* that are automatically copied along side the appropriate HTML files during the HTML generation process. These CSS files are:

```
reposRoot/repository-detail.css
reposRoot/html/package-detail.css
```

6 Design and extension notes

The basic idea is that using the VIEWS file and the known repository structure (location of packages and extracted vignettes), we represent the details for each package in the repository in a *PackageDetail-class* instance.

packageDetail-class objects know how to write themselves to HTML using the `htmlValue` method. We used the *XML* package's `xmlOutputDOM` function to build up the HTML documents. Each HTML producing class extends *Htmlized-class* which contains a slot to hold the DOM tree and provides a place to put methods that are not specific to any given HTML outputting class.

In terms of extending this to generate the biocViews, have a look at `setDependsOnMeImportsMeSugges`
which builds up an adjacency matrix representing package dependencies, importations, and suggestions. The matrix is square with rows and columns labeled with the names of the packages. The entries are 0/1 with $a_{ij} = 1$ meaning that package j depends on package i.

6.1 Details on HTML generation

I started by breaking the `htmlValue` method for *PackageDetail-class* into one helper function for each logical section of the HTML we produce (author, description, details, downloads, and vignettes). That made the long method short enough to be readable.

In order to be able to mix and match the different chunks and be able to more easily create new renderings, it seemed that it would be easiest to be able to render to HTML each chunk with a method. One possibility is a function `htmlChunk(object, descriptions)` where the dispatch would be done using a switch statement or similar.

A more flexible approach is to create dummy classes for each output "chunk". Each dummy class contains (subclasses) *PackageDescription* and that's it. We then can take advantage of the behavior of the `as` method to convert.

```
> ## Define classes like this for each logical document chunk
> setClass("pdAuthorMaintainerInfo", contains="PackageDetail")
> setClass("pdVignetteInfo", contains="PackageDetail")
> ## Then define a htmlValue method
> setMethod("htmlValue", signature(object="pdDescriptionInfo"),
+           function(object) {
+               node <- xmlNode("p", cleanText(object@Description),
+                               attrs=c(class="description"))
+               node
+           })
> ## Then you can make use of all this...
> ## Assume object contains a PackageDetail instance
> authorInfo <- as(object, "pdAuthorMaintainerInfo")
> dom$addNode(htmlValue(authorInfo))
>
```

One advantage of this setup is that we can now define a method to generate complete HTML documents that will work for all the dummy classes. Hence mix and match.

6.2 A note on the htmlValue method for PackageDetail

We could parameterize as follows. Not sure this makes things easier to follow, but it does demonstrate how you could start building up documents in a more programatic fashion.

```
details <- list(heading=list(tag="h3", text="Details"),
                content="pdDetailsInfo")
downloads <- list(heading=list(tag="h3", text="Download Package"),
                  content="pdDownloadInfo")
vignettes <- list(heading=list(tag="h3",
                    text="Vignettes (Documentation)"),
                  content="pdVignetteInfo")

doSection <- function(sec) {
    dom$addTag(sec$heading$tag, sec$heading$text)
    secObj <- as(object, sec$content)
    dom$addNode(htmlValue(secObj))
}

lapply(list(details, downloads, vignettes), doSection)
```