RKWard plugin development with the `rkwarddev` package

m.eik michalke

December 13, 2011

Writing plugins for `RKWard` means writing at least two XML files (a GUI description and a plugin map), one JavaScript file (to create the `R` code), maybe a help file (again in XML), and for plugins who should be distributed, a DESCRIPTION file. Furtermore, all of these files need to be in a certain directory structure.

The `rkwarddev` package aims to simplify this, by enabling you to fulfill all the listed tasks in just one `R` script.

1 About the package

You might ask why you should write R scripts to generate plugins, if you could just directly write the XML and JavaScript files. First of all, you don't have to use this package at all, it's totally fine to code your plugins how ever you like. The main reason why I wrote this package is that I like to really concentrate on what I'm doing, so this is my attempt to avoid the need to switch between several files in three different languages all the time. I wanted to be able to constantly "think in `R`" while working on a plugin, and to oversee everything that matters in one script. As a side effect, a lot of useful automation was implemented, so using this package will definitely save you quite some amount of typing.

2 Before we start

It is important to undertsand that while `rkwarddev` can help you to make designing new plugins much easier, you still need to know how the generated XML and JavaScript files work and interact. That is, if you didn't yet read the *Introduction to Writing Plugins for RKWard,*[1] please do so before you start working with this package. Once you're sure you understand how plugins in `RKWard` actually work, just come back here.

[1] `http://rkward.sourceforge.net/documents/devel/plugins/index.html`

3 Ingredients

If you look at the contents of the package, you might feel a little lost because of the number of functions. So let's first see that there's actually some order in the chaos.

Most functions start with the prefix `rk.` to indicate that they somehow belong to `RKWard` (we'll get to the exceptions later). After that, many names have another abbreviation by which they can roughly be classified into their specific "area" of plugin development:

- `rk.XML.*()`: XML code for GUI description (and plugin maps)
- `rk.JS.*()`: JavaScript code
- `rk.rkh.*()`: XML code for help pages

In short, you should find a `rk.XML.*()` equivalent to every XML tag explained in the *Introduction to Writing Plugins for RKWard*,[2] e. g. `rk.XML.dropdown()` to generate a `<dropdown>` menu node. There are a few functions for JavaScript generation which fall out of this scheme. That is because firstly they should be intuitively to use just like their JavaScript equivalent (like `echo()`), and secondly they are likely to be used very often in a script, so short names seemed to be a blessing here (like `id()` or `tf()`).

Adding to that, there are some special functions, which will all be explained later, but here's the list, roughly ordered by the development stage they're used for:

- `rk.paste.JS()`: Paste JavaScript code from `rkwarddev` objects
- `rk.XML.plugin()`: Combine XML objects into one plugin GUI object
- `rk.JS.scan()`: Scan a GUI XML file (or `rkwarddev` object) and generate JavaScript code (define all relevant variables)
- `rk.JS.saveobj()`: Scan a GUI XML file (or `rkwarddev` object) and generate JavaScript code (save result objects)
- `rk.JS.doc()`: Combine JavaScript parts into one plugin JavaScript file object
- `rk.rkh.scan()`: Scan a GUI XML file (or `rkwarddev` object) and generate a help page skeleton
- `rk.rkh.doc()`: Combine XML objects into one help page object
- `rk.plugin.component()`: Combine XML, JavaScript and help file objects into one plugin component object (i. e. one dialog, so *one* plugin can provide *several* dialogs in one package)
- `rk.testsuite.doc()`: Paste a testsuite skeleton
- `rk.XML.pluginmap()`: Combine XML objects into one plugin map object

[2] `http://rkward.sourceforge.net/documents/devel/plugins/index.html`

- `rk.plugin.skeleton()`: Generate actual plugin files from the component, test-suite and plugin map objects (i. e., put all of the above together)
- `rk.build.plugin()`: Compress the generated files into an installable `R` package for distribution

3.1 Exceptions to the rule

As said before, there are some functions that fall out of the explained name scheme,i. e. they don't start with `rk.<XML|JS|rkh>.*()`. They are all relevant for the generation of JavaScript code, and this is just a short overview, how you use them will also be explained later on:

- `echo()`: Produces an equivalent of the JavaScript `echo()` function
- `id()`: Similar to paste, but replaces `rkwarddev` objects with their ID value
- `ite()`: Short for "**i**f, **t**hen, **e**lse", a shortcut to generate JavaScript `if() {} else {}` conditions
- `qp()`: Short for "**q**uote & **p**lus", like `id()`, but with different replacement defaults
- `tf()`: Short for "**t**rue/**f**alse", a shortcut to `ite()` for XML checkbox objects
- `rk.comment()`: Creates a comment object to show up in the generated code – works for both XML and JavaScript generation

4 Writing a plugin

The previously mentioned *Introduction to Writing Plugins for RKWard*[3] has a chapter on `rkwarddev` as well, which also includes a full example plugin already. This section will not so much repeat what you can learn there, but rather explain the basic steps to create a plugin "the `rkwarddev` way" in general. While doing that, we'll explore some of the alternative options you have when using different functions.

Some of them might not be so obvious at first, but I believe that once you know them, you'll learn to like them, too.

To begin with some background info, this package makes use of another `R` package I wrote, called `XiMpLe`[4]. It is a *very* simple XML parser/generator, hence its name. All `rkwarddev` functions dealing with XML utilize tools of this package, that is, the XML objects created are most likely of class `XiMpLe.node`, if not some other `XiMpLe` class.[5]

[3] `http://rkward.sourceforge.net/documents/devel/plugins/index.html`
[4] `http://reaktanz.de/?c=hacking&s=XiMpLe`
[5] The machanism for JavaScript is basically the same, but those classes and tools are all part of `rkwarddev` itself.

4.1 What you see is what you get, in the end

Both packages also come with `show` methods for their objects. This means that the object you create and how it looks when called in an R session are not the same: What you will see in your terminal is what the object *would* look like if you *pasted* it to a file, using the `paste` functions of the packages:

```
> rk.XML.frame(label="Example XML object")

<frame label="Example XML object" id="frm_ExmplXML">
</frame>
```

If you examine the actual structure of the created object with `str()`, you can see the gory details:

```
> str(rk.XML.frame(label="Example XML object"))

Formal class 'XiMpLe.node' [package "XiMpLe"] with 4 slots
  ..@ name      : chr "frame"
  ..@ attributes:List of 2
  .. ..$ label: chr "Example XML object"
  .. ..$ id   : chr "frm_ExmplXML"
  ..@ children  : list()
  ..@ value     : chr ""
```

Most of the time, you won't ever have to worry about that, since the objects will be handled by the package functions automatically. But it's important to understand that the results of these functions aren't simple character strings, allthough it might look like it at a first glance.

4.2 Generating XML code

In the section before, we have already generated our first XML object: the `rkwarddev` function `rk.XML.frame()` produced a `<frame>` node. I guess this is pretty straight forward. Usually, a frame needs some content nodes to make sense, so we'll now create two simple checkbox[6] objects, put them inside the frame and look at the result:

```
> myCheckbox <- rk.XML.cbox(label="Check me!")
> myCheckbox2 <- rk.XML.cbox(label="No, check me!!!", chk=TRUE)
> myFrame <- rk.XML.frame(myCheckbox, myCheckbox2, label="Example XML object")
> myFrame

<frame label="Example XML object" id="frm_ExmplXML">
        <checkbox id="chc_Checkme" label="Check me!" value="true" />
        <checkbox id="chc_Nocheckm" label="No, check me!!!" value="true" checked="true" />
</frame>
```

[6]As an almost unique exception, the name of `rk.XML.cbox()` does not match the name of the generated XML node, "checkbox“. Don't worry about that.

What we can see here is that the generated code will automatically be indented, so the result will not only work, but still be human readable and look nice (and probably even better than what some might come up with otherwise...). We can also learn how nodes are made nested children of other nodes: All `rkwarddev` functions which can create parent to more than one node have the special ”dots“ parameter in their signature (`...`). That way you can give them arbitrary numbers of XML objects, and they just know what you want them to do with them.

4.2.1 IDs

If you have a closer look you can also see one of the packages' automatic features: The node objects automatically received ID values, allthough we didn't specify any. By default, allmost all functions supporting IDs have `id.name="auto"` set, which as we've seen will not cause the ID to become `"auto"`, but a generated value. Usually an auto-ID is combined of the abbreviated node type and the abbreviated label given. So here, our `<frame>` node labelled ”Example XML object“ got the ID `frm_ExmplXML`. If we wanted a node to have some specific ID, we can use the `id.name` argument:

```
> rk.XML.cbox(label="Check me!", id.name="specificID")

<checkbox id="specificID" label="Check me!" value="true" />
```

Now, the fact that these nodes are actually objects of class `XiMpLe.node` gives us direct access to their attributes, including the ID:

```
> myCheckbox <- rk.XML.cbox(label="Check me!")
> myCheckbox@attributes[["id"]]

[1] "chc_Checkme"
```

Again, mere mortals probably won't use this directly, but this makes it easy for functions read the IDs of XML nodes and use them. For example, if you wanted to define a varselector and a varslot, so the latter can take objects from the former, you need to give the varselector an ID and define that as the source in the varslot. If you wrote XML directly, you would give the `source` attribute the actual ID. With this package, you *can* do that too, but there is a much more elegant solution: Give the whole XML object and let the function extract the ID itself:

```
> (myVarselector <- rk.XML.varselector(id.name="my_vars"))

<varselector id="my_vars" />

> (myVars <- rk.XML.varslot(label="Chose a variable", source=myVarselector))

<varslot id="vrsl_Chosvrbl" label="Chose a variable" source="my_vars" />
```

So basically you define an XML object and then re-use this single object throughout your plugin script, be it for actual XML generation or, as in this case, only for getting its ID. This means, in other words, you can tell the varslot ”take variables from this object“, you don’t have to worry about IDs *at all.* Just remember how you named an object and you can do all kinds of things with it.

This context dependent object handling will become even more useful when we get to the JavaScript part.