gWidgetsWWW

John Verzani, gWidgets@gmail.com

December 18, 2010

Abstract:

The **gWidgets** package provides an API to abstract the interface for a few of the available GUI toolkits avaiilable through R. The **gWidgetsWWW** package provides an implementation of the **gWidgets** API for use with through the web. Using just R commands, interactive web GUIs can be produced.

The current status of the project is still experimental. It should work with most modern browsers, although some widgets (gcanvas, gsvg) are not working equally for all browsers. (Some of these require the `canvas` tag to be implemented.)

1 Overview

This package provides both a means to layout a web page to serve as an interactive GUI, but also a means to have an `R` session called from the GUI to manipulate the web session.

There are two choices for the webserver:

- For local use (no outside internet access) the dyanmic help web server of R is leveraged. To use this, the function `localServerStart` is called to start the server, or `localServerOpen` is used to load a file. The command

  ```
  localServerOpen("/Examples/ex-simple-gui.R", package="gWidgetsWWW")
  ```

  will open an example page.

- One can serve web pages to a wider audience through the RApache package `http://biostat.mc.vanderbilt.edu/rapache/` for R. This embeds an R process within the Apache web server. This requires some configuration. The defaults work for a standard Ubuntu installation.

A modern set of web tools includes three primary domains: HTML for markup of the text, CSS for the layout of the page, and JavaScript to manipulate the page. This package requires none of these, as it leverages a popular javascript library (ext from `www.sencha.com`) to provide a consistent look and the javascript interactivity. As the pages are not centered around displaying content, but rather interactive controls, there is no need for HTML knowledge.

To make a GUI in **gWidgets** requires two steps: writing a script and calling that script.

A simple script could look like:

```
w <- gwindow("simple GUI with one button", visible=FALSE)
g <- ggroup(cont = w)
b <- gbutton("click me", cont = g, handler = function(h,...) {
  gmessage("hello world", parent = b)
})
visible(w) <- TRUE
```

To call this script can be done various ways. Suppose we store it in the file "script.R". If we want to run it through the local web server we would open it with a command like:

```
localServerOpen("full_path_to_script.R")
```

Whereas if we wanted to serve it throug rapache, we would place the file into an appropriate directory (this is configured when RApache is configured for gWidgetsWWW). The default is /var/www/gWidgetsWWW. Then the script is called from the browser with a url such as

```
http://your.domain.com/gWidgetsWWWrun/script.R
```

The domain is yours, the "gWidgetsWWWrun" is the default URL to execute a script, but this can be configured to your likeing.

GUIs made with gWidgetsWWW are not as snappy as other web GUIs. The main reason for this is the fact that callbacks to manipulate the page are sent back into R (the server) and then returned. In many GUIs, this is avoided by using javascript directly on the web page. The clear tradeoff is ease of programming for the R user (R not javascript) versus speed for the user.

2 Top level windows

Web GUIs are different than desktop GUIs. Not only are they slower, as they have lag time between the GUI and the server, there can only be one top-level window.

In a script we have a few restrictions on top-level windows, not present in the **gWidgets** API. Namely: as just metioned, only one top-level instance per script; this must also be stored in a global variable; this object is not visible until explicity made so through a call to `visible` and this is done only after the GUI is layed out.

Subwindows are possible. These are other `gwindow` instances using the `parent` argument to specify an object that acts as the parent of a subwindow. (an animation will appear from there, say.)

3 The containers

The **gWidgetsWWW** package has all the following containers:the top-level container `gwindow`, subwindows also constructed through `gwindow` (use a parent object); the box containers`ggroup`, `gframe` and `gexpandgroup`; the tabular layout container `glayout`; the notebook container `gnotebook`, but no `gpanedgroup`.

To make a component appear in response to some action – such as happens with gexpandgroup, one can add it to a box container dynamically. Or one can put it in a `ggroup` instance and toggle that containers visibility with the `visible` method, in a manner identical to how `gexpandgroup` is used.

4 The widgets

Most – but not all – of the standard widgets work as expected. This includes labels (`glabel`), buttons (`gbutton`), radio buttons (`gradio`), checkboxes (`gcheckbox`, `gcheckboxgroup`), comboboxes (`gcombobox`), sliders (`gslider`), spinboxes (well kind of) (`gspinbutton`), single-line edit boxes (`gedit`), multi-line text areas (`gtext`), dataframe viewers (`gtable`) and editors (`gdf`), images (`gimage` – the image is a url), menu bars (`gmenu` – but not toolbars), statusbars (`gstatusbar`).

The dialogs (except `gbasicdialog`) are implemented.

The widgets `galert`, `ghtml` and `gaction` are all implemented.

The `gtree` widget is implemented, but only has one column for display – not a grid.

The `gbigtable` widget is an alternate to `gtable` allowing for large data frames to be displayed. This widget has a paging feature.

The widget gcommandline is implemented for local versions only – running arbitrary commands is a *huge* security risk.

No attempt has been made to include the compound widgets gvarbrowser, ggenericwidget, gdfnotebook, ggraphicsnotebook.

4.1 graphics

There is no plot device avaiable. Rather, one can use the **png** device driver, say, to create graphic files which are then shown using gimage. The function getStaticTempfile should be used to produce a file, as this file will sit in a directory that can be accessed through a URL. This URL is returned by convertStaticFileToUrl. Other drivers may be used, but for server use they should not depend on X11.

This is modified from the example ex-image.R

```
> fileName <- getStaticTmpFile(ext=".png")
> png(file=fileName, width=500, height=500) # pixel size
> print(histogram(rnorm(100)))
> dev.off()
> gimage(convertStaticFileToUrl(fileName), cont=g)
```

Two package-specific widgets, gcanvas and gsvg, can be used for displaying non-interactive graphics files through the **canvas** device or the RSVGTipsDevice (the SVGAnnotation device should work as well). There are used similarly: You create the widget, you create a file. The graphics device writes to the file (similar to the png device driver, say). This file can be assigned to the widget at construction time, or later through the svalue method. For gsvg the file must be accessible as a URL, so the getStaticTempfile function should be used. The **canvas** device uses a newer HTML entity, canvas, which is not supported on all browsers. The gsvg package uses a SVG (scalable vector graphics) format. This format again has some issues with browsers, but seemingly fewer. The **RSVGTipsDevice** device has simple features for adding tooltips and URLs to mouse events. The **SVGAnnotation** package allows this and much more.

4.2 Quirks

A number of little quirks are present, that are not present with other **gWidgets** implementations:

Top-level windows As mentioned above, top level windows must be global to the script. There can be only one and it must be named. (A seach over the global variables with a given class is used to find the toplevel window in a handler).

The top level window is not made visible at first. (A good idea in any case, but not the default for `gwindow`. To create a window the `visible` method, as in `visible(w) = TRUE`, is used. (When this is issued the javascript to make the page is generated.)

Environment of handlers The handlers are run in an environment that does not remember the loading of any packages beyond the base packages and **gWidgetsWWW**. So in each handler, any external packages must be loaded.

Working quietly The use of the options `quietly=TRUE` *and* `warn=FALSE` should be used with `require` when loading in external packages. Otherwise, these messages will be interpreted by the web server and an error will usually occur.

Debugging woes Debugging can be tough as the R session is not readily available. The error messages in the browser are useful. For development of the package the firebug (`getfirebug.com`) add on to FireFox has proven very useful. Alternatively, it can be helpful to source in the script at the command line to see if there are errors.

4.3 Data persistence

AJAX technologies are used to prevent a page load every time a request is made of the server, but Each time a page is loaded a new R session is loaded. Any variables stored in a previous are forgotten. To keep data persistent across pages, one can load and write data to a file or a data base.

4.4 Comboboxes

The `gcombobox` example shows how comboboxes can show a vector of items, or a data frame with a column indicating an icon, or even a third icon with a tooltip. As well, the `gedit` widget does not have a type-ahead feature, but the combobox can be used for this purpose.

The following will set this up.

```
> cb <- gcombobox(state.name, editable=TRUE, cont = w)
> cb$..hideTrigger <- TRUE ## set property before being rendered
```

This package is implemented independently of the **gWidgets** package, and so there may be some unintended inconsistencies in the arguments. The package uses the **proto** package for object-oriented support, not S3 or S4 classes. There are some

advantages to this, and some drawbacks. One advantage is the user can modify objects or call their internal methods.

5 Installation

The local server is installed with the package. Just type `localServerStart` to begin, although this won't load a file. For that try `localServerStart("Examples/ex-index.R", package='gWidgetsWWW')`.

Installation of the RApache server requires several steps:

1. Install the RApache module. For Ubuntu or Debian machines there is a copy and paste script on the RApache.net website

2. Install the **gWidgetsWWW** package from CRAN. This must be installed so that the webserver can run it.

3. The **gWidgetsWWW** has a `RApache-gWidgetsWWW.conf` file that must be installed and configured. To install it with UBUNTU something like this will work (provided you have permission)

   ```
   > system(sprintf("cp %s /etc/apache2/conf.d",
           system.file("templates/RApache-gWidgetsWWW.conf", package="gWidgetsWWW"
   ```

   To configure the `RApache-gWidgetsWWW.conf` file requires a few things:

   - Specifying where the `mod_R` apache module is. The default is for an Ubuntu installation
   - Specifying where to find scripts to run through `gWidgetsWWWrun`. The default is the set of examples, but this should be changed to the directory (or directories) where you wish to put the gWidgetsWWW scripts. The directory need not part of the htdocs directory.
   - Or specifying where to find brew scripts **gWidgetsWWW**. (This is needed for creating graphics files.)
   - Some other options have reasonable defaults, although many have a hard coded path to the gWidgetsWWW package in them. If this is not a standard place for an Ubuntu install these will need to be changed to that given by:

     ```
     > system.file("", package="gWidgetsWWW")
     ```

6 Security

Security – is a big deal. Web servers can be hacked, and if hacked the hacker has full access to the server. This can be scary. The local server blocks all requests that are not to the local IP 127.0.0.1, preventing outside access. As for outside access, although it is not believed that RApache is any less secure than other Apache modules, you can protect yourself by running the entire setup within a virtual machine. There is easy to install, reasonably priced (or free) commercial software from VMWare (`www.vmware.com`). For open-source fans, the VirtualBox project (`www.virtualbox.org`) also has software. One can install this, then run the author provided appliance. Or one can install the virtual software, install a host OS (ubuntu linux say), then install Apache and R then RApache etc. It actually isn't so hard to do.

The call from the web server back into RApache can also be source of insecurity. The **gWidgetsWWW** package allows only a limited number of calls back from a web page, which should in theory be secure. But if the script is not secure, there is nothing the package can do. Scripts must *never* trust that data sent from the web page to the server is safe. It should be coerced into any desired format, and never evaluated. Using `eval` say allows any one to run R commands on the server which given the power of R means they have full control of the server.

The web server communicates back to the web browser through an AJAX call. This is supposed to be secure, as only javascript code that originates from the same server as the initial page is executed.