Abstract:

The **gWidgets** package provides a programming interface (an API) to write graphical user interfaces from within R. The API trades off power for relative simplicity and has proven useful for writing small to moderately complicated GUIs. The **gWidgetsWWW2** package (`https://github.com/jverzani/gWidgetsWWW2`) implements much of the API for programming GUIs for the web. It uses the Ext JS JavaScript libraries ((`www.sencha.com`) to provide the toolkit and the **Rook** package of Jeffrey Horner to integrate with `R`'s local web server. The `nginx` web proxy can be used to serve pages to a public audience.

As the name suggests, this is the second version of the implementation. This one uses the newer Ext JS 4.1 libraries and internally uses `R`'s reference classes – not the **proto** package – to provide mutable objects.

1 Overview

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. In addition, often there is a back end language, such as `python`, for server-side programming. The combined set of pre-requisites can be daunting, though clearly many have succeeded in mastering them. The goal of this package is for the R programmer to avoid nearly all of these considerations, yet still be able to produce interactive web sites powered by `R`.

This package leverages a popular javascript library (ext from `www.sencha.com`) to provide a consistent look (the CSS part) and the javascript interactivity. `R` is used for the "server-side" programming, and `R`'s internal web server is used for serving pages, through the **Rook** package of Jeffrey Horner.

The primary contribution of the package is a) a means to create JavaScript from `R` code (from the **gWidgets** API) and b) a means to easily incorporate this into a web page. These web pages are not centered around displaying content, but rather for creating interactive controls. However, if desired, the package can be used to integrate in with an existing web page, such as those provided through **brew** templates.

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

A simple script creating an interactive GUI could be as simple as:

```
> w <- gwindow("simple GUI with one button", visible=FALSE)
```

```
DEBUG only
```

```
> g <- ggroup(cont = w)
> b <- gbutton("click me", cont = g, handler = function(h,...) {
+   gmessage("hello world", parent = b)
+ })
```

(Though you'd be unimpressed with the layout of the widgets.)

If called correctly, the package translates these commands into JavaScript commands to manipulate a web page:

```
> out <- w$dump()
> head(out)

$`1`
[1] "var ogWidget_ID1 = new Ext.container.Viewport({'id':'gWidget_ID1','layout':'fit'

$`2`
[1] "var ogWidget_ID1_toplevel=ogWidget_ID1;"

$`3`
[1] "var ogWidget_ID1=ogWidget_ID1.child(0);"

$`4`
[1] "document.title='simple GUI with one button';"

$`5`
[1] "var ogWidget_ID2 = new Ext.panel.Panel({'id':'gWidget_ID2','border':false,'hideB

$`6`
[1] "ogWidget_ID1.add(ogWidget_ID2);"
```

The easiest way to see this webpage is to run the script through a specific URL which is created when the `load_app` command is executed. For example, if the script above is stored as `hello.R`, then this call will open the page and load the url `http://127.0.0.1:PORT/custom/HelloApp/indexgw.rhtml`:

```
> load_app("hello.R", app_name="HelloApp")
```

Other means to incorporate this into a web page are possible, but require additional work.

Not only does the GUI get constructed, the package provides a means to callback into the R process from the web server to further manipulate the page. The "handler" in the definition of the button, b above, is called through an AJAX call when the button is pressed. In this case, the gmessage call produces additional JavaScript that causes a modal message dialog to be produced. The handlers are written in R – not JavaScript. This makes them easier to write, but slower to process, as their use requires a round trip to the server from the web page. This process of communication between client and server happens more often than this, as the state of the GUI is synchronized with the R process, as changes are made.

Documentation for the package is provided in help pages, but the main **gWidgets** API is better documented in that package. (Though that package is not a dependency and should not be loaded, as this package is a stand alone implementation.) There are numerous package-specific features beyond the basic **gWidgets** API. Mostly these are implemented as reference-class methods and are documented with the contstructors.

The documentation provides a few basic uses, more complicated examples are provided through the demo.

2 Top level windows

Web GUIs with **gWidgetsWWW2** 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. This is a gwindow instance called without a parent argument.

Other gwindow instances are either a) subwindows (appearing as a separate window) if the argument renderTo is not specified of b) rendered to the DOM element with id specified by renderTo. [1]

Common to all windows (and child widgets) is a single instance of the GWidgetsTopLevel class. In order to share this, each widget constructor requires a specification of the widget heirarchy through a parent container (with the container) or if the widget is not in the heirarchy, a parent argument. This toplevel instance is stores references to each created object, process the outgoing JavaScript queue, and is used to route incoming callback requests. The callbacks are all evaluated within an enviroment that is also stored within the toplevel object. As the toplevel instance is created each time a page is loaded, this evaluation environment is not persistent.

[1] The latter allows **gWidgetsWWW2** to integrate with other web pages. Within a page, leave a div tag with an id some_id, say, and then pass the argument renderTo="some_id" to the constructor. The ex-multiple example demonstrates.

3 The containers

The **gWidgetsWWW2** package implements the basic widgets of the **gWidgets** APIL:

- the top-level container `gwindow` and subwindows also constructed through `gwindow` (use a parent argument);
- the box containers`ggroup`, `gframe` and `gexpandgroup`;
- tabular layout container `glayout`;
- the notebook container `gnotebook`;
- the split-window container `gpanedgroup`.

In addition to these, the `ExtJS` libraries make it easy to provides some other containers:

gstackwidget This widget is similar to a notebook container, but without the tabs. It is part of the the **gWidgets2** API (`https://github.com/jverzani/gWidgets2`). It is useful here, as each new page load creates its own unique evaluation environment, so page loads do not share global variables. To work around this, one can use this widget to flip between "pages."

gborderlayout A common web-layout is a "border" layout with 5 areas to place components: a "center", and 4 satellites: north, south, east and west. As with `gexpandgroup`, a user can resize the space allocated to each area.

gpanel The gpanel widget is a container for other JavaScript libraries. Basically it creates a `DIV` tag, which can be overwritten by external JavaScript calls. An example shows how the `d3` JavaScript libraries (`http://mbostock.github.com/d3/`) can be incorporated.

As with **gWidgets**, child components can be added and deleted from parent containers dynamically. The `gexpandgroup` widget can be used to hide or disclose parts of GUI.

The size of containers in **gWidgetsWWW2** is different from other **gWidgets** implementations. When used to produce a stand-alone app (the basic usage), the top-level window takes up the entire web browser screen. Its child component will is allocated this entire space. Box containers have the property (similar to **RGtk2**) that child components expand to fill the space orthogonal to the direction of packing (vertical boxes and child components that take up the maximum horizontal space). Thus in the simple "hello world" example, the horizontal box container fills the entire page, and the button stretches vertically to fill that space – a really tall button. A common use then is to use nested containers with different packing directions:

```
> w <- gwindow("simple GUI with one button", visible=FALSE)
```

```
DEBUG only
```

```
> g <- ggroup(cont = w, horizontal=FALSE, use.scrollwindow=TRUE)
> button_group <- ggroup(cont=g, horizontal=TRUE) ## opposite to g
> b <- gbutton("click me", cont = button_group, handler = function(h,...) {
+   gmessage("hello world", parent = b)
+ })
```

The `use.scrollwindow=TRUE` call will allow the GUI to be larger than the allocated screen space, often a useful thing.

Containers also have the odd property, that they may have no dimension allocated to them, despite having children. You may need to specify a height or width. In **gWidgetsWWW2** the constructors all have arguments `width` and `height` for specifying the initial width and height. These take values in numbers of pixels. The `size` assignment method can also take values in percentages of allowed space, as in `100%`.

4 The widgets

The standard widgets of **gWidgets** are implemented:

- Buttons and labels with `gbutton` and `glabel`;
- The widgets to select from a vector of values: `gcheckbox`, `gcheckboxgroup`, `gradio`, `gcomobox`;
- The widgets to select from a range of values: `gslider` and `gspinbutton`;
- The text widgets `gedit`, for single-line edit boxes and `gtext`, for multi-line text areas;
- The table selection widget `gtable`;
- The tree widget `gtree`

- The data frame editor `gdf`;
- Images can be viewd through `gimage` – the image is a url
- menu bars but not toolbars with `gmenu` and statusbars (`gstatusbar`);
- File uploads can be incorporated through `gfile`.

The basic dialogs are implemented except `gbasicdialog`). These include `galert`, `gmessage`, `ginput`, and `gcalendar`. All require a `parent` argument.

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

The `ghelp` constructor is not provided. The **helpr** package is very useful for that.

In addition to the above, the package has other widgets. In addition to the different widgets for graphics described below, there is:

`ghtml` A widget for displaying HTML either specified as a string or as a URL.

`ggooglemaps` A widget for displaying google maps, described below.

4.1 graphics

There is no *interactive* plot device. Rather one can use a variety of non-interactive devices and an accompanying widget.

These include:

- the **canvas** device along with the `gcanvas` widget. This device writes out JavaScript commands and uses an underlying canvas object on the web page. This requires an HTML5 compliant browser. There is some support for mouse events.
- The `gsvg` widget can be used to display svg graphics, as produced by the `svg` driver or the **RSVGTipsDevice** driver.
- The `gimage` widget can be used to display graphics produced by the `png` driver, among others.

Both the last two require both a file and a url, the file to write to, the url to use by the browser. If the file is created by `get_tempfile` the details are implemented by the widget.

The `canvas` device is used like a non-interactive device (open the device, create the graphic, call `dev.off()`). For example,

```
> w <- gwindow("Display a graphic with canvas")

DEBUG only

> width <- height <- 400
> ## make graphic
> f <- tempfile()
> canvas(f, width=width, height=height)
> hist(rnorm(100))
> dev.off()

null device
          1

> ## display graphic
> g <- ggroup(cont = w, horizontal=FALSE)
> gcanvas(f, width=width, height=height, container=g)

An object of class GCanvas
```

The `gsvg` widget is similar, though one uses a file that can be served as a url, so replace `tempfile` with the package-provided `get_tempfile`, as with `get_tempfile(ext='.svg')`. The advantage of `svg` graphics is they scale to fill the space of their container, unlike the canvas object.

The `gcanvas` device allows access to HTML5's underlying canvas tag methods, allowing one to manipulate objects in the widget. The widget itself can responds to mouse clicks through a handler specified with `addHandlerClicked`. The "h" argument is a list with additional components `x` and `y` (containing the ndc coordinates of the point with $(0, 0)$ being the lower left corner) and `X` and `Y` (containing the pixel coordinates of the click, with $(0, 0)$ being the upper left corner. The `x` and `y` values can be converted into `user` coordinates through `grconvertXY`, but there is a catch – the device needs to be reopened with the same dimensions, as that information is lost when the device is closed. See the example `ex-gcanvas` for an illustration.

4.2 ggooglemaps

The `ggooglemaps` widget provides access to a sliver of the google maps API. See the help page for an illustration.

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.

The `JsonRPCServer` class provides an alternative. This allows the web page to call back into objects in the `R` processes global workspace by method name. The help page for `json_rpc_server` shows how this can be used for a page counter.

5 Additional details

There are several places where additional methods are provided by reference class methods beyond the basic **gWidgets** API.

To step back, each constructor produces an instance of some reference class. These instances have their own methods defined for them. The **gWidgets** API simply dispatches to the appropriately named method. (For example, `svalue` looks for `get_value` and `get_index`, whereas the assignment method looks for `set_value` or `set_index`.) Reference class methods are called using `R`'s object oriented notation.

5.1 Toplevel object

The toplevel object is stored in the `toplevel` property of the objects and can be accessed as follows:

```
> w$toplevel
```

This is shared among the objects:

```
> identical(w$toplevel, g$toplevel)
```

```
[1] TRUE
```

The `toplevel` instance holds a reference to the request that made the page. This can be useful to get script information (not useful here though, as Sweave isn't calling as Rook does):

```
> w$toplevel$the_request$path_info()
```

5.2 The toplevel window

The toplevel object holds a queue of JavaScript commands to send back to the browser. One can add to this queue, if desired. For convenience, each component has the method `add_js_queue` to do so. One specifies a string of JavaScript to passback, for example:

```
> w$add_js_queue("alert('hello world');")
```

```
[1] "55"
```

Now when the page makes a request for the server, this will also be passed back.

By design, the server only responds to requests from the clients. One can not push server requests to clients (as can be done with web sockets). If such a thing is desired, the top-level window has the method `start_comet` which causes the page to poll the server periodically, and not just in an event-driven manner.

The toplevel window can be used to get and set cookies. The methods `cookies` returns the cookies available when the page is created, the method `set_cookie` can set a cookie. (A cookie set with `set_cookie` will not appear in the `cookies` list, as these are from when the page is created.)

5.3 Ext

The underlying constructors and handlers basically do just one thing: write out JavaScript code for the `ExtJS` libraries to interpret at the browser. Some features allow one to do this directly, if desired.

First, the constructors have the argument `ext.args` to specify additional arguments to the `Ext` constructor. These are specified through a named list and the package will do a conversion to a JavaScript object.

Next, the objects have a few methods. The `call_Ext` method takes a method name and other arguments and calls the corresponding `Ext` method. The arguments are coerced from `R` objects. For example, if `b` is a `gbutton` instance, then `b$call_Ext("setVisible", FALSE)` will adjust the visibility of the widget. (As will the call `visible(b) = FALSE`, in this example.). The method `ext_apply` will merge configuration values from list.

5.4 Debugging

Finding bugs can be tedious, as they can appear in different places:

R issues Issues with R code can be detected b trying to run the script at the R console (`source`, say). One can also run the script locally, and see if any errors are logged at the console.

JavaScript issuse These are more subtle. Most browsers have some JavaScript debugging tools built in, or easily added (e.g., firebug for Mozilla). The debugger for Chrome allows one to see what is requested and what the response is.

It is important to realize that just looking at the source will not help. The source for a script loaded with `load_app` simply loads some external packages and then issues a call to populate the page. This call does not appear in the page source, but rather is a response to an AJAX request. The JavaScript debugger can show you this.

6 Installation

The local usage is configured when the package is installed. After installation typing `demo(gWidgetsWWW2)` should open a page allowing the examples to be run. The `load_app` function can be used to turn a script into a web page.

6.1 Serving pages on the internet

At this point the package can be intergrated in with **nginx**, say, to serve pages to the public. The **nginx** web server is basically a proxy, routing requests to the underlying **Rook** process.

> **No claim is made that this is industrial grade**. For that, use a real web-development stack. This should be able to handle moderate usage, though there are many areas where scaling will just plain fail.

A basic configuration for **nginx** simple reroutes calls to the Rook process. The `load_app` call opens up port 9000 by default, so a configuration in `nginx.conf` like:

```
location /custom {
  proxy_pass http://localhost:9000/custom;
}
```

will route all urls `http://you.domain.com/custom/XXX` to the `Rook` process. Jeffrey Horner has more details in a gist at `https://gist.github.com/6d09536d871c1a648a84`.

Each app enabled by `load_app` has an option for an "`app_name`" and this maps to the URL: `http://you.domain.com/custom/app_name/indexgw.rhtml`.