gWidgetsWWW: Creating Interactive Web Pages within **R**

John Verzani
CUNY/College of Staten Island

Abstract

The **gWidgetsWWW** package provides a framework for easily developing interactive web pages from within R. It uses the API developed in the **gWidgets** programming interface to specify the layout of the controls and the relationships between them. The web pages may be served locally under R's built-in web server for help pages or from an **rApache**-enabled web server.

Keywords: GUI, R.

1. Introduction

The **gWidgets** package (Verzani 2007) provides a programming interface (API) for R (R Development Core Team 2012) users to several different toolkits for producing user interfaces (GUIs). The API trades off the full power of the underlying toolkits, for a simplified but often sufficient set of GUI features that the R user can quickly learn. The **gWidgetsWWW** package implements much of the **gWidgets** API for developing interactive web applications programmed with R scripts.

There is a long history of integrating R with web browsers. For example, the `browseEnv` function, which generates a web page summarizing the objects in the global environment, has been in base R at least since version 1.9. A quick glance at the web interfaces part of R's frequently asked questions (R-FAQ, Hornik 2010) shows at least a dozen projects. However, web frameworks have evolved quite rapidly and many listed on the FAQ are now somewhat obsolete or no longer maintained.[1] The **gWidgetsWWW** package provides another

---

[1]The **gWidgetsWWW** package will be no exception. Currently a **gWidgetsWWW2** package is on GitHub, http://www.github.com, and will replace this one. This new package provides a nearly identical user interface, but underneath integrates with the **Rook** (Horner 2012) package for better integration with R's internal web server and uses reference classes, not the **proto** (Kates and Petzoldt 2011) package.

framework for the R user to create interactive web pages, in this case using relatively modern, yet mature technologies. This package is available from the Comprehensive R Archive Network at http://CRAN.R-project.org/package=gWidgetsWWW.

A number of recent examples blend web interfaces with R in new ways, allowing R to be run remotely over the internet. Most prominently, **RStudio** (**RStudio** 2012) is a complete development environment for R built using industry standard web technologies (even the desktop version). For developing interfaces, the **rApache** project (Horner 2010) provides a module that embeds R within the Apache web server. In Oems (2010) are several beautifully implemented R-based web applications using **rApache**. The **Rwui** application (Newton and Wernisch 2007) provides a web interface to easily create basic applications powered by R within a relatively complicated Apache setup (using Tomcat). Other means to integrate R with web technologies include **Rserve** (Urbanek 2003); **rpy2** (Gautier 2011), to interface Python web frameworks with R; and RevoDeployR (Revolution Analytics 2012), which provides an API allowing R to be used as a back-end engine within a web programming framework. Finally, we mention that Tibco's **Spotfire** (TIBCO Software Inc. 2012) application provides a facility to easily produce dynamic web dashboards, although not directly a solution for the R user.

As of recent versions of R, a built-in web server has been provided for serving dynamically generated help pages locally. This local server can be repurposed to integrate interactive web pages with a user's R session. For example, The **helpr** package (Wickham and Schloerke 2010) uses this framework to provide a much enhanced help page displayer with "live" examples. The **googleVis** package (Gesmann and de Castillo 2011) can use the help-page server to interface R with Google's visualization tools. More recently, the **Rook** package has been developed and provides standard specifications for writing web applications for any web server, including R's built-in one.

A typical web programming environment mixes several technologies such as cascading style sheets for styling the content and markup languages, typically HTML or XML, for providing content – R provides a number of packages, such as **hwriter** (Pau 2010), for producing HTML from R objects. Once one moves beyond static html files for web pages, templates are often used. Within R, the combination of **rApache** and **brew** (Horner 2011) makes an able templating framework. The **R.rsp** (Bengtsson 2012) package provides another alternative. More widely used frameworks are often based around PHP or Python (e.g., **django**, Django Software Foundation 2010). All template solutions require the web server to call the underlying template engine.

To add interactivity to a web page, JavaScript is typically used for creating dynamic effects client side. JavaScript is executed by the browser and allows a programmer to manipulate a web page's content. For more complicated uses, standard JavaScript libraries are often used. One such is Ext JS (Sencha Inc. 2012). The use of JavaScript allows for AJAX technologies which have changed the way web applications are now programed. AJAX, unlike CGI, allows asynchronous communication between a client and web server without interfering with the display of the existing page unless requested.

A typical web framework then consists of numerous different technologies, a mastery of which may be beyond the R programmer's desire. For applications defined through the **gWidgets** API, the **gWidgetsWWW** package hides these technologies from the R user, so that creating interactive web applications can be done with only a knowledge of R. For this convenience, the package trades off the power, flexibility, speed, and scalability of other web technologies.
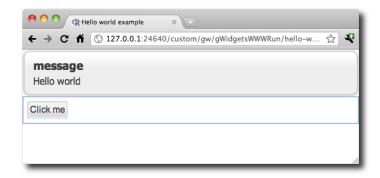
Figure 1: Screenshot of "Hello World" example. The interface consists of a lone button, which when clicked initiates the displayed message to appear.

The package uses the internal help-page server for local use and integrates in with **rApache** for serving pages to a wider audience.

2. Usages

We begin with a simple "Hello World" example. The following script demonstrates the basics needed to produce an interactive web page (Figure 1):

```
w <- gwindow("Hello world example")
g <- ggroup(container = w)
b <- gbutton("Click me", container = g)
addHandlerClicked(b, handler = function(h, ...) {
  galert("Hello world", parent = w)
})
visible(w) <- TRUE
```

The `gwindow` constructor maps to a web page, the text specifying the page's title. To this a container (`ggroup`) is added and then a control (`gbutton`) added to that. (The `container` argument specifies the parent container of a new widget.) This completes the basic layout of this simple application. Afterwards, the interactivity is added by assigning a handler to the event of clicking the button through the `addHandlerClicked` method. When the button is clicked, a call to produce a dialog is initiated. This involves the browser making an AJAX call back to the server, which then calls R, which in turn responds with JavaScript commands instructing the web browser how to display the dialog. The last line of the script shows a method call for the `gwindow` instance, in this case, one that causes the JavaScript commands to be output to the web browser to instruct the browser to draw the page.

To run this script depends on whether the script is run locally or served through **rApache**. If done remotely, then the script is placed in an appropriate directory and invoked through a browser by a URL that reflects the script's filename. For local use, the script's filename is simply passed to the function `localServerOpen` which calls `browseURL` with the appropriate URL.[2]

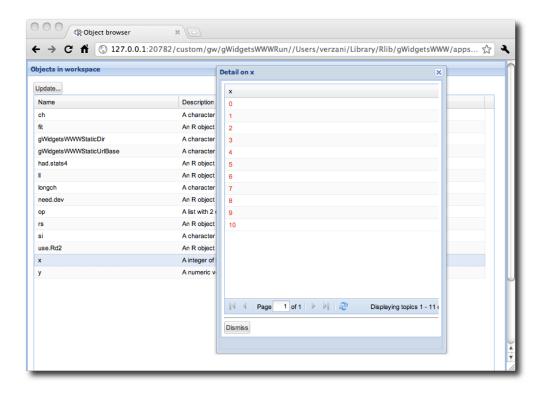[2]For **gWidgetsWWW2** the local script is run through `load_app`.

Figure 2: Screenshot of object browser created by `gw_browseEnv`, an alternative to `browseEnv`.

We present in the following a few examples and note that the package itself has a number of examples and a package vignette. In addition, most basic **gWidgets** scripts can be run with little or no modification.

2.1. An object browser

First to show how the package can be used to supplement features provided with a package, we discuss a script used to enhance a user's local command-line experience. In **gWidgetsWWW** we provide the function `gw_browseEnv` as an alternate to `browseEnv`:

```
R> library("gWidgetsWWW")
R> gw_browseEnv()
```

This function just calls `localServerOpen` on a script that comes with the package. The main GUI looks like Figure 2. Using the local help server allows such displays to be interactive, unlike `browseEnv`, and the **gWidgetsWWW** package makes them easy to create. For example, the code to create the table of items is basically:

```
 workspace_objects <- gtable(list_objects(), container = g)
```

where `list_objects` creates a data frame of the objects in the workspace using a few generic functions to give different levels of detail on an object. More detail is provided in a subwindow after clicking on an item. For example, clicking on a matrix or data frame object dispatches to:
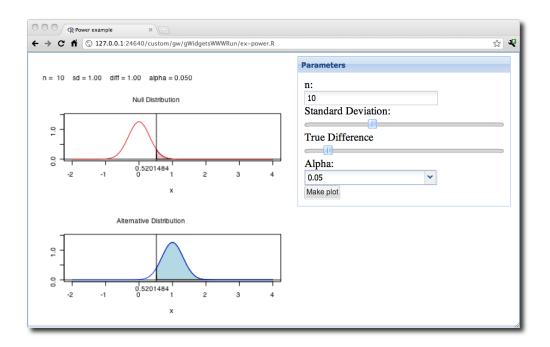
Figure 3: Screenshot of a web application for displaying a power calculator.

```
detailOn.matrix <- function(x, varname, cont, width, height, ...) {
  g <- ggroup(container = cont, horizontal = FALSE)
  glabel(sprintf("Detail on %s", varname), container = cont)
  gbigtable(as.data.frame(x), container = g, width = width, height = height)
}
```

The `gbigtable` widget allows one to display very large data sets without having to push all the data from the R process to the web browser at once.[3]

2.2. Using images in a GUI

For R there are several packages providing interfaces to underlying graphical toolkits (e.g., **RGtk2**, **rJava**, **qtbase**). However, the most widely used is the **tctlk** package. This popularity can not be attributed to the inherent ease or power of using the underlying Tk toolkit, but rather the fact that installation of the toolkit is nearly ubiquitous, as the libraries are shipped with the base Windows installation of R. The **gWidgetsWWW** package provides an easy to install alternative, as no external libraries are needed to create an interface. Once the package is installed, one can then provide additional functionality, say to students, via scripts on the web.

As an example, the following script (see Figure 3) will create a power demonstration modeled after one in the **TeachingDemos** (Snow 2012) package. The script also illustrates the use of the **canvas** package, which provides a graphic device for R that produces JavaScript to manipulate a canvas element in an HTML5-ready browser. This allows us to update our graphic in response to changes of the controls. The `our.power.examp` call below is to a function, not

---

[3]Paging is part of `gtable` in **gWidgetsWWW2**, so `gtable` can be used in place of `gbigtable`.

shown, from the **TeachingDemos** package, only slightly modified to avoid the calls to `legend`, which are not supported by the `canvas` device.

First, the layout of the canvas and primary controls:

```
require("canvas")
width <- height <- 400

w <- gwindow("Power example")
g <- ggroup(container = w)
cnv <- gcanvas(width = width, height = height, container = g)

f <- gframe("Parameters", container = g, horizontal = FALSE)
glabel("n:", container  =f)
n <- gedit(10, container = f)

glabel("Standard Deviation:", container = f)
stdev <- gslider(5, 20, 1, value = 10, container  = f)

glabel("True Difference:", container = f)
diff <- gslider(0, 10, by = 1, value = 1, container = f)

glabel("Alpha:", container = f)
vals <- seq(.05, .25, by = 0.05)
alpha <- gcombobox(vals, editable = TRUE, container = f,
  coerce.with = as.numeric)

b <- gbutton("Make plot", container = f)
```

The following function to produce a plot is called to make the initial graphic and then added as a callback to each of the controls so that the graphic is updated when there is a change, suitably interpreted.

```
plotIt <- function(...) {
  f <- tempfile()
  canvas(f, width = width, height = height)
  our.power.examp(svalue(n), svalue(stdev) / 10,
            svalue(diff), svalue(alpha))
  dev.off()
  svalue(cnv) <- f
}
plotIt()
sapply(list(n, stdev, diff, alpha, b), addHandlerChanged, handler = plotIt)
visible(w) <- TRUE
```

The `gcanvas` widget is specific to **gWidgetsWWW**. There are other alternatives, such as `gsvg` and `gimage`, for displaying plots produced by R's device drivers.

2.3. Remotely serving files

The previous examples emphasized the use of the package to run local web applications. For this, no further configuration beyond installing the package is required. However, to allow a script to run remotely does require additional configuration.

For serving pages remotely **rApache** is used. The **rApache** project creates a module for the Apache web server that embeds an R process within the server. This greatly speeds up the overhead of calling on R through a system call. Although **rApache** can be used with other UNIX environments, it is easily installed within an Ubuntu distribution using the standard installation procedures and a custom repository.

Once installed, Apache must be configured for the **rApache** module. Afterwards, configuration for **gWidgetsWWW** is also needed. The package comes with a standard configuration template that may be used. In addition to defining several variables employed by the R process in the **rApache** model, this template:

- Specifies a base URL, with default `gWidgetsWWWrun`, that directs the server to return a **gWidgetsWWW** script.

- Specifies one or more directories for **rApache** to look for such scripts.

- Specifies directories and URLs for static files such as image files, and

- Specifies the URL for any AJAX calls.

For an Ubuntu installation, the defaults may be used. In this case, one simply places a script, say `ex-poll.R`, into the directory `/var/lib/gWidgetsWWW` and calls the file through the URL http://domain.name/gWidgetsWWWrun/ex-poll.R.

We illustrate with a simple application to take a poll on the favorite state of the users. Figure 4 shows the web application after a few users have voiced their opinion.

By design, **gWidgetsWWW** keeps a separate environment for each web page it creates. This allows concurrent requests to the same page. It also means we need a means to store persistent data used by a script besides the global environment of the R process. In the following we simply keep a text file with the results, one line per user. In order to keep multiple users from simultaneously writing to this file, we use a lock file. We begin by defining two script-global variables:

```
resultsFile <- "/tmp/results.txt"
lockFile <- "/tmp/results.txt.locked"
```

Next, we create two functions to interact with the lock file:

```
getLock <- function(lockfile, ctr = 10) {
  while(ctr > 0) {
    if(file.exists(lockfile)) {
      ctr <- ctr - 1
    } else {
      message("locked", file=lockfile)
      return(TRUE)
```

```
    }
  }
  return(FALSE)
}
clearLock <- function(lockfile)
  if(file.exists(lockfile))
    file.remove(lockfile)
```

For this GUI, we use a radio button group for a selector and a table to display our data. The following code defines a data frame to hold our survey data and then creates the layout:

```
Survey.data <- data.frame(State = state.name[1:4], vals = rep(0,4),
  stringsAsFactors = FALSE)
w <- gwindow("Survey results")
gstatusbar("Powered by gWidgetsWWW", container = w)
g <- ggroup(container = w, horizontal = TRUE)
g1 <- gframe("Select your favorite state:", container = g,
  horizontal = FALSE)
rb <- gradio(Survey.data[,1], container = g1)   # main control
gbutton("vote", container = g1, handler = function(h,...) {
  updateResults(svalue(rb))
})
g2 <- gframe("Results:", container = g, width = 300)
results <- gtable(Survey.data, cont = g2)
size(results) <- c(300, 150)
```

The button handler stores the newly selected value and then updates the table display, `results`. It must check first to see if it can actually write to the file, and if denied, a message is produced.

```
updateResults <- function(newValue) {
  if(!file.exists(resultsFile))
    file.create(resultsFile)

  if(getLock(lockFile)) {
    if(!missing(newValue))
      cat(newValue, "\n", file = resultsFile, append = TRUE)
    x <- scan(resultsFile, what = "character", quiet = TRUE)
    clearLock(lockFile)
    if(length(x))
      Survey.data$vals <- sapply(Survey.data[,1], function(i) sum(x == i))
    results[] <- Survey.data
  } else {
    galert("Try again, resource is busy.", parent = w)
  }
}
```

The script is finished by updating the current results then calling `visible<-` to write out the JavaScript.
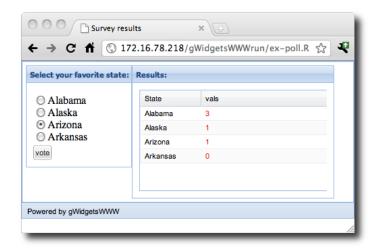
Figure 4: The poll application after a few users have voted. The basic layout includes two framed containers, a radio button group, and a table widget.

```
updateResults()
visible(w) <- TRUE
```

Poll results, such as the above, are usually embedded within a larger web page. The **gWidgetsWWW** scripts create single web pages, though. However, one can use an `iframe` tag to display a script within a frame on a page.

3. Implementation

The **gWidgets** API is not described here, but we do note that this package introduces some slight changes to that API. The basic idea of **gWidgetsWWW** is the script defining a primary `gwindow` object which will map to a web page. When this main window is "displayed" the package writes JavaScript commands to draw the GUI. The JavaScript uses the open-source version of the Ext JS libraries for convenience. The setup puts some restriction on the scripts, as the JavaScript is created by calling the `visible<-` method of the main window. As well, for interactivity, when a widget has a "handler" some JavaScript is inserted that calls back to the R session to run the handler. The handler call returns JavaScript to manipulate the page. The design of this process requires the lone top-level window to be a *global* variable in the script.[4]

It is worth noting that while the browser initiates calls to the R process which then returns a response, as of now, there is no means to initiate a call to the browser from the R process. (For example, in the workspace browser example there is an "Update..." button needed to refresh the web application with the current workspace variables.) This causes a problem. When a callback is processed with R it needs to be aware of the values the widgets are storing at the browser and it has no means to request these. As such, changes to a control are propagated back to the R process through an AJAX call each time the control is updated. Not only does

---

[4]This is not necessary in **gWidgetsWWW2**, where one only needs to provide a single parentless `gwindow` instance.

this synchronization depend on a reliable internet connection, there is an assumption that these asynchronous calls have happened prior to the callback being processed.

The underlying code is based on the **proto** package. In writing **gWidgetsWWW** scripts there are a few places where **proto** methods can be used to extend the features provided by the base **gWidgets** API. The help page `?"gWidgetsWWW-package"` provides some details.

Web GUIs, while ubiquitous now, have tradeoffs when compared to desktop GUIs. For **gWidgets** a major tradeoff is the difficulty of debugging errors during script writing. Developing under the local server and a good `JavaScript` debugger, such as `firebug` for Firefox (Resig 2012) or the built-in one for `Chrome`, proves very helpful.

Additionally, having R provide the dynamic aspects comes with a tradeoff. The benefit is the R programmer need not know any HTML, `CSS` or `JavaScript` to write dynamic web pages. The cost is paid in speed (as all actions must go from the browser to R and back) and scalability. The latter is a problem as each page creates a session which is stored as the entire R environment. The size of which can potentially be quite large if one is not careful.

Finally, server applications must defend against maliciously entered user input. The **gWidgets** controls, for the most part, are secure as all user-passed values are coerced to certain types of values (integer indices if possible, or logical values say). However, text-based entries can not be so sanitized. One should be very careful when evaluating or displaying any commands that may have come from a user

4. Conclusion

The **gWidgetsWWW** package provides an alternative means to write interactive web GUIs that, at the expense of scalability and flexibility, greatly simplifies the necessary web programming skills for the R user.

Acknowledgments

The author would like to thank the very helpful comments of the editor and referees in the preparation of this manuscript.

References

Bengtsson H (2012). ***R.rsp**: Dynamic Generation of Scientific Reports.* R package version 0.7-5, URL http://CRAN.R-project.org/package=R.rsp.

Django Software Foundation (2010). ***django**: The Web Framework for Perfectionists.* URL http://www.djangoproject.com/.

Gautier L (2011). "**rpy2**: A Simple and Efficient Access to R from Python." URL http://rpy.sourceforge.net/rpy2.html.

Gesmann M, de Castillo D (2011). "**googleVis**: Interface between R and the Google Visualisation API." *The R Journal*, **3**(2), 40–44. URL http://journal.R-project.org/.

Horner J (2010). *rApache: Web Application Development with R and Apache.* R package version 1.1.20, URL http://www.rapache.net/.

Horner J (2011). *brew: Templating Framework for Report Generation.* R package version 1.0-6, URL http://CRAN.R-project.org/package=brew.

Horner J (2012). *Rook: A Web Server Interface for R.* R package version 2.0-4, URL http://cran.r-project.org/package=Rook.

Hornik K (2010). "The R FAQ." ISBN 3-900051-08-9, URL http://CRAN.R-project.org/doc/FAQ/R-FAQ.html.

Kates L, Petzoldt T (2011). *proto: Prototype object-based programming.* R package version 0.3-9.2, URL http://CRAN.R-project.org/package=proto.

Newton R, Wernisch L (2007). "**Rwui**: A Web Application to Create User Friendly Web Interfaces for R Scripts." *R News*, **7**(2), 32–35. URL http://CRAN.R-project.org/doc/Rnews/.

Oems J (2010). "Interactive Web Applications." URL http://www.stat.ucla.edu/~jeroen/.

Pau G (2010). *hwriter: HTML Writer – Outputs R Objects in HTML Format.* R package version 1.3, URL http://CRAN.R-project.org/package=hwriter.

RStudio (2012). "**RStudio**." URL http://RStudio.org/.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Resig J (2012). "**firebug**: Web Development Evolved." URL http://getfirebug.com/.

Revolution Analytics (2012). "RevoDeployR." URL http://www.revolutionanalytics.com/products/enterprise-deployment.php.

Sencha Inc (2012). "Ext JS 4.1 JavaScript Framework for Rich Apps in Every Browser." URL http://www.sencha.com/products/extjs/.

Snow G (2012). *TeachingDemos: Demonstrations for Teaching and Learning.* R package version 2.8, URL http://CRAN.R-project.org/package=TeachingDemos.

TIBCO Software Inc (2012). "**Spotfire**." Version 4.5, URL http://spotfire.tibco.com/.

Urbanek S (2003). "**Rserve** – A Fast Way to Provide R Functionality to Applications." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna, Austria. ISSN 1609-395X. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/.

Verzani J (2007). "An Introduction to **gWidgets**." *R News*, **7**(3), 26–33. URL http://CRAN.R-project.org/doc/Rnews/.

Wickham H, Schloerke B (2010). **helpr**: *Help for R*. R package version 0.1.2.2, URL http://CRAN.R-project.org/package=helpr.

Affiliation:

John Verzani
Department of Mathematics
College of Staten Island
2800 Victory Boulevard
Staten Island, NY 10314, United States of America
E-mail: verzani@math.csi.cuny.edu
URL: http://www.math.csi.cuny.edu/verzani/