Using ReportingTools within Shiny Applications

Gabriel Becker and Jessica L. Larson

October 13, 2014

Contents

1 Introduction

A primary strength of `ReportingTools` is that it provides powerful, customizable facilities for creating rich, interactive (sortable, filterable, pagable, etc.) and aesthetically pleasing HTML tables based on many disparate types of `R` objects. `shiny` is a Web application framework developed by RStudio, Inc. which allows the creation, and deployment of Web applications using only `R` code. Often these Web applications involve the display of `R` objects or output, but formatting and rendering of complex `R` objects is not the focus of the `shiny` framework.

Using the techniques in this vignette, `ReportingTools`' formatting and display capabilities, including both the default mechanisms and the full range of customizable behavior, can be incorporated into `shiny` applications, allowing the creation of powerful Web applications which involve the display of `R` objects representing complex data and analysis results. This vignette assumes knowledge of the `shiny` framework. Readers who are not familiar with shiny are encouraged to read the official `shiny` tutorial here before continuing.

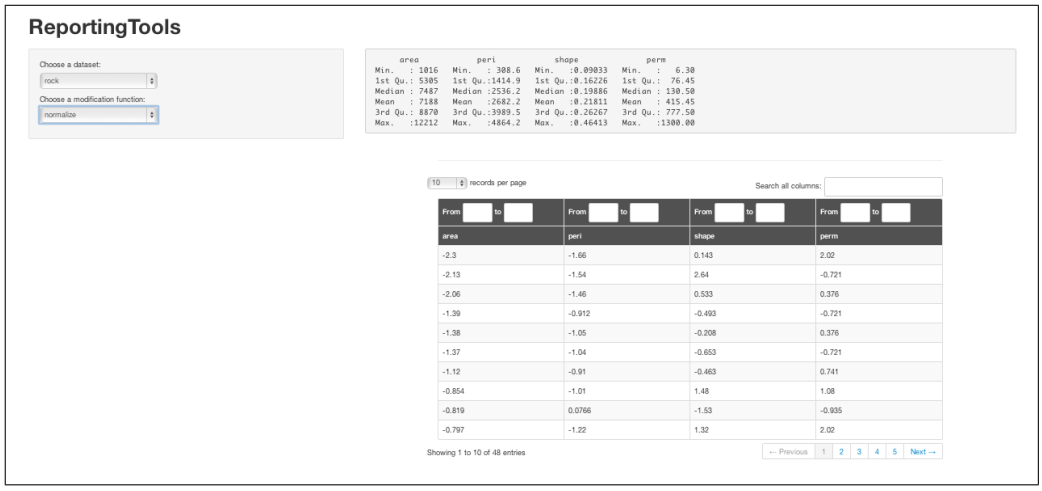

Figure 1: A shiny web application which uses `ReportingTools` to display R `data.frame` objects

The example we will discuss in this document, pictured above, gives the viewer the opportunity to choose between three data frames and displays both a summary and a ReportingTools-powered table containing the chosen data. We will discuss in detail only portions of the code specific to the interface between `shiny` and `ReportingTools`. Full code for the application is included in our package in the inst/examples/shinyexample directory. To run the example, copy the `inst/examples/shinyexample/Ui.R` and `inst/examples/shinyexample/server.R` files to your working directory run the following from an R session:

```
> library(shiny)
> myRunApp()
```

2 Changes to ui.R when using ReportingTools

The single largest change to a ui.R file in order to add `ReportingTools` functionality is that `ReportingTools`' JavaScript and CSS files must be included in the header of the resulting page so that the `ReportingTools` tables function properly.

We define a function `custHeaderPanel` function which accepts the `title` and `windowTitle` arguments accepted by `shiny`'s `headerPanel` function but also accepts additional arguments `js` and `css`. These are ex-

pected to be character vectors which specify locations of additional Javascript and CSS libraries, respectively. These files are then read and inserted into the header as code in <script> and <style> tags, respectively.

With this function defined we are able to use it within the standard `shiny` page layout functions, such as `pageWithSidebar`, in place of the `headerPanel` function. In particular, we include all (Javascript) files in extdata/jslib and all Twitter Bootstrap based CSS files in extdata/csslib:

```
>     custHeaderPanel("ReportingTools",
+                     js = list.files(system.file("extdata/jslib", package="ReportingTools"),
+                                     full.names=TRUE),
+                     css = list.files(system.file("extdata/csslib", package="ReportingTools"),
+                       pattern="bootstrap", full.names=TRUE),
+                     )
```

These Javascript and CSS files will be included in the header of the resulting dynamic HTML page, allowing our `ReportingTools`-based output to behave correctly.

Code for specifying input controls is identical whether or not `ReportingTools` is being used to format the output and is omitted here.

Finally, output elements which will be formatted by `ReportingTools` should be declared as `htmlOutput`. We do this for the `view2` element in the code below:

```
>    mainPanel(
+                 verbatimTextOutput("summary"),
+                 htmlOutput("view2")
+                 )
```

This indicates to the `shiny` system that the output will be HTML code ready to be inserted directly into the specified element. With this our page layout is defined and we are ready to write the server.R code which will populate it.

3 Changes to server.R when using ReportingTools

Our task here is to specify a rendering function which can interface with the `ReportingTools` publish mechanism. To do this we first create a Report (within server.R, outside of any function calls) with `ReportHandlers` created via the `shinyHandlers` constructor:

```
> myrep = HTMLReport(reportDirectory = "./",shortName="bigtest",
+   handlers = shinyHandlers)
```

These `ReportHandlers` will stream the HTML form of any elements added to our Report directly to Rout (the same as the default destination of cat, and one used heavily by `shiny` to collate output).

We then use (or define) a custom rendering function, `renderRepTools`. By using this custom rendering mechanism and `ReportHandlers` combination, shiny is able to "hear" elements being added to our report and insert them into the dynamic HTML of our Web App.

To make use of this we simply publish elements to our report within the expression passed to `renderRepTools`:

```
> ###use RT to display output
>   output$view2 <- renderRepTools({
+     publish(datasetInput(), htmlrep, .modifyDF = modifyInput())
+   })
```

The resulting web application is controlled entirely by `shiny`, but has the added rendering power built into `ReportingTools`. Though we used a standard `data.frame` in this example, we can expand this application to more general biological data and `Bioconductor` objects which would be difficult to effectively display without `ReportingTools`. Furthermore, all customization mechanisms for the HTML output discussed in the other vignettes are fully functional in this setting.

4 References

Huntley, M.A., Larson, J.L., Chaivorapol, C., Becker, G., Lawrence, M., Hackney, J.A., and J.S. Kaminker. (2013). ReportingTools: an automated results processing and presentation toolkit for high throughput genomic analyses. *Bioinformatics*. **29**(24): 3220-3221.