

# BrowserViz

Paul Shannon

April 25, 2015

## Contents

---

1	Introduction	1
2	A Standard Message Format	2
3	The Simple BrowserViz “Application”	3
4	Simple Subclassing Overview	3

## 1 Introduction

---

*BrowserViz* provides the basis for, and a very simple working example of, interactive R/browser visualization. Thus two interactive powerful and complementary environments are linked, creating a powerful hybrid setting for exploratory data analysis.

This work is motivated by our belief that contemporary web browsers, supporting HTML5 and Canvas, and running increasingly powerful Javascript libraries (for example, *d3* and *cytoscape.js* have become the best setting in which to develop interactive graphics for exploratory data analysis. We predict that web browsers, already powerful and easily programmed, will steadily improve in rendering power and interactivity, and thus remain the optimal setting for interactive R visualization for years to come. (An example of what the future of browser-based graphics may hold can be seen in Google’s *NaCL* project <https://developer.chrome.com/native-client>, a sand-boxing technique for running native OpenGL 3D applications in the browser.)

*Shiny* and *htmlwidgets* are two very popular packages which provide solutions to this same general problem: how does one use the power of web browser graphics from R? Both of these package create bindings in R to HTML widgets and Javascript objects. This creates representations for these objects in R: a button, a *d3* scatterplot, an interactive geographic map. The two packages provide elegant support for these bindings and a clear path to creating more of them.

*BrowserViz’s* difference from these fine and popular packages by provding only *loose coupling* of R and the browser. In slogan form, our approach can be summarized, “let R be R, and Javascript be Javascript”. Two rich programming environments are linked, but the environments are kept maximally ignorant of each other. Only simple JSON messages pass back and forth, and these are at a high semantic level: no HTML, CSS or Javascript. Rather than creating representations of web objects in R, which would be tight-coupling of the two environments, *BrowserViz* provides a style of programming in which

- Web objects are (just as in standard widespread web programming) created in HTML, CSS and Javascript
- JSON and websockets provide simple explicit message passing between R and the browser
- High-level web objects (i.e., an xy plotter, or a network viewer) are created by and manipulated by Javascript functions at the request of high-level R function calls
- Web elements can initiate (call back to) R functions
- A traditional event-driven architecture is used throughout, in which events are either R function calls or user (keyboard or mouse) actions in the browser.

*BrowserViz* provides a very low threshold for those wishing to create R/web browser visualizations. This base package hides the complexity of websocket initiation and message passing. The websocket communication channel is created with a single R function call. Passing messages and handling responses is similarly simple. The intricacy (or simplicity) of the web browser interface is determined by the programmer. A vast collection of easily available books, examples, tutorials, and support websites make web browser programming especially easy to learn. The *BrowserViz* approach will be of interest to any programmer interested in the visualization of data, and proficient in – or willing to learn – both R and Javascript.

Standalone web sites can be created, but the primary intended audience for this package is the R programmer exploring and analyzing data in R and using the browser visualization for the indispensable benefits it provides. We hope that many visualization tools will be created. We provide a simple x-y plotter (see *BrowserVizDemo*) to illustrate how to write a BrowserViz subclass application. The *RCyjs* is a full-featured visualization tool for network visualization built upon *cytoscape.js* (see <http://js.cytoscape.org>). and the web browser.

The *BrowserViz* class, though a base class intended for subclassing, includes a simple demo which performs a few elementary browser manipulations, and queries the browser for some simple state (window size, window title, browser version). The principal goal of the package is to provide the the websocket “plumbing” along with a standard (simple, open-ended) message protocol for communicating between the two environments.

## 2 A Standard Message Format

---

Just as the ubiquitous and language-neutral *websockets* protocol provides the *BrowserViz* communication mechanism, so does *JSON* provide the message notation. Native data types in R (a named list) and Javascript (an object, with key:value pairs) are easily converted to and from JSON by libraries standard in each language. We have adopted a simple, adaptable data structure flexible enough for all of the uses so far encountered. In JSON (and Javascript):

```
{cmd: "setBrowserWindowTitle", status: "request", callback:"handleResponse",
  payload: "BrowserViz Demo"}}
```

Websocket servers both send and receive messages. Thus a typical *BrowserViz* event begins with sending a message from one environment to the other, and often concludes with some sort of a return or “callback” message.

- *cmd*: the name of the operation the sender wishes to be performed by the receiver.
- *status*: might be “success”, “failure”, “error”, “deferred response”.
- *callback*: provided by the sender, this specifies the operation which the receiver is to call *in the client* after it (the receiver) completes the operation it was asked to perform.
- *payload*: An open-ended data structure, sometimes empty, as simple as a character string, as complex as any conceivable deeply nested list.

```
> library(jsonlite)
> msg <- toJSON(list(cmd="setBrowserWindowTitle", status="request",
+                   callback="handleResponse", payload="BrowserViz demo"))
```

The callback for this request could be empty, which by convention we encode as the empty string. The calling code, in R, and the receiving code, in Javascript, only need to be consistent. If the caller provides a non-empty *callback*, the Javascript receiver should craft and send a return message with the canonical four fields specifying *cmd=callback* and any *payload* the caller expects, perhaps

```
{cmd: "handleResponse", status: "success", callback:"", payload: "BrowserViz Demo"}
```

An empty *payload* could also be used, in which case the *success* status of the return command is the only information returned from Javascript to R. All decisions of this sort are left to the programmer. Often the same person writes the R and the Javascript code that talk back and forth over the websocket. If different programmers are involved, then careful communication and documentation is required, of the expectations, constraints and *payload* structure.

### 3 The Simple BrowserViz “Application”

---

We predict that the principal use of *BrowserViz* will be as a base class for other rich visualization packages, and that authors of those derived classes will be able to proceed without any direct involvement in the nuts and bolts of websocket creation and handling. Nonetheless, *BrowserViz* is a complete R/browser application, albeit one with only a few features. These features (R methods on the BrowserViz object), few though they be, are automatically available to all *BrowserViz* subclasses.

- *port*:
- *ready*:
- *browserResponseReady*:
- *getBrowserResponse*:
- *closeWebSocket*:
- *send*:
- *getBrowserWindowSize*:
- *getBrowserWindowTitle*:
- *setBrowserWindowTitle*:

This describes the R component of the package. It is complemented by an HTML/Javascript/CSS component; call it the “web component”. In simple visualizations this web component is a single file whose name is provided to the BrowserViz constructor. We depend upon the *httpuv* package: it functions primarily as a websocket server, but also includes (and starts up as) a simple http web server. Once the web socket is created and opened by *BrowserViz*, the constructor asks the default web browser to display the web component’s html page. Javascript in this page then opens a connection back to the *BrowserViz* web socket server, and the application is ready to go, driven by user events – either function calls in R, or mouse and keyboard events in the browser.

### 4 Simple Subclassing Overview

---

The *BrowserVizDemo* package shows how to build upon *BrowserViz*. *BrowserVizDemo* is an S4 class; its Javascript component uses the *d3* graphics library to render x,y points onto an auto-scaled canvas with labeled axes. The class contains *BrowserViz* and thus has all of the above operations, and these in addition

- *plot*: takes an x and y vector as arguments
- *getSelection*: returns the names of all d3 selected points, in the browser plot, to R.

Thus the R programming is quite simple.

Neither Javascript nor HTML provide any native mechanism for class inheritance. So in order to create the web component of a new visualization application, we provide a simple Javascript module, *BrowserViz.js* available (for now; a better location will be found) at

<http://oncoscape.sttrcancer.org/oncoscape/js/BrowserViz.js>

It simplifies your Javascript coding significantly. It is used in your webapp (Javascript, HTML, CSS file/s) like this (see *BrowserVizDemo* for the full story).

```
hub = BrowserViz();           // create an instance of the BrowserViz hub
bvDemo = BrowserVizDemo()    // create your own Javascript module object
bvDemo.init(hub);            // initialization of your module
hub.addOnDocumentReadyFunction(bvDemo.initializUI); // register functions for the hub to call

// register your web apps message handlers: the hub will dispatch
// incoming messages to these, based on the "cmd" field in the JSON message
hub.addMessageHandler("ready", ready)
hub.addMessageHandler("plotxy", d3plotPrep)
```

```
hub.addMessageHandler("getSelection", getSelection)

// with all setup complete, now
hub.init();           // socket connections setup
hub.start();          // and started

// your handlers (locally defined functions which do things in the browser)
// will sometimes (or often) send a response back to R
// the hub makes that easy.
// first, build up your four-field JSON message, then
hub.send(return_msg);
```

A close study of the [BrowserVizDemo](#) will give you a good start towards creating your own custom webapp and BrowserViz subclass.