Building R Web Applications with Rook

Jeffrey Horner

Rook is a web server interface for R web applications so that they can see the web server in a consistent way, regardless of where they are deployed.

The R Rook package provides convenience software that implements the Rook interface.

Since R 2.13, you've be able to run your R web applications via Rook on your own desktop using the internal R web server.

Motivation (and some history)

- 2005 rApache presented at Directions in Statistical Computing.

```
handler <- function(r){
    apache.write(r,"<h1>Hello World!</h1>")
    OK
}
```

- 2007 created brew.

```
<h1>Hello <%=$GET['name']%>!</h1>
```

- rApache and brew provide a capable platform for developing and deploying R web applications.

Motivation (and some history)

- October 2009 R 2.10 released and includes internal web server (Rhttpd) for displaying help files in browser.
- Oct/Nov 2009 Hadley Wickam starts sinartra and helpr, technology to replace existing R help system. Uses the internal R web server.
- Late 2010 Jeffrey Horner creates an interface layer in rApache that mimicks Rhttpd to run sinartra apps (experimental).
- January 2011 Simon Urbaneks adds access to web request headers for handlers, the missing piece for a near-complete evironment for web app development in R.

Motivation

- rApache and Rhttpd provide different interface to web developers.
- What to do?
- Look for how others have solved the problem.
- Python, Ruby, and Perl created web server independent interfaces for their applications. Python created PWSGI first, Ruby created Rack with inspiration from WSGI, and Perl created Plack and was inspired by both.
- Rack is the simplest.

BUT WHY BUILD WEB APPS WITH R

- I'm not a statistician; I'm a tool maker and I like programming and sticking things together like legos ;)
- Maybe you want to build an interactive demonstration of your model for all to use.
- Maybe you want to build an RPC stat server and use that as a back-end to your front-end RoR, PHP, or Python web app. (OpenCPU)
- Or maybe you're just weird.
- There's far worse ideas out there, like programming an editor in Lisp.
- I kid.

Rook is a specification first.

A Rook application is an R function that takes exactly one argument, an environment, and returns a list with three named elements: the 'status' , the 'headers', and the 'body'.

```
helloworld <- function(env){
     list(
          status=200,
          headers = list(
               'Content-Type' = 'text/html'
          ),
          body = paste('<h1>Hello World!</h1>')
     )
}
```

status is an HTTP status value and must be greater than or equal to 100. **headers** is a named list that contains string values only corresponding to valid HTTP headers. **body** is either a character or raw vector. If the character vector is named with value 'file' then value of the vector is interpreted as the location of a file, the contents of which are served as the response.

```
helloworld <- function(env){
    list(
        status=200,
        headers = list(
            'Content-Type' = 'text/html'
        ),
        body = paste('<h1>Hello World!</h1>')
    )
}
```

Rook is a package of convenience, too.

```
helloworld <- function(env){
    req <- Request$new(env)
    res <- Response$new()
    res$write('<h1>Hello World!</h1>\n')
    res$finish()
}
```

Request and Response are reference classes.

Rook is Rackable and Powerful.

```
app <- Builder$new(
    Static$new(
        urls = c('/css','/images','/javascript'),
        root = '.'
    ),
    Static$new(urls='/plots',root=tempdir()),
    Brewery$new(url='/brew',root='.'),
    App$new(function(env) {
        req <- Request$new(env)
        res <- Response$new()
        res$redirect(req$to_url('/brew/useR2007.rhtml'))
        res$finish()
    })
)
```

Static serves static files matching urls. Brewery calls brew on files that match urls. If the url does not match, then the last app redirects the browser to '/brew/useR2007.rhtml'.

Reference Classes are Easy

```
Foo <- setRefClass(
    'Foo',
    fields = c('fee','fi'),
    methods = list(
        initialize = function(fee='fee',fi='fi',...){
            fee <<- fee
            fi <<- fi
            callSuper(...)
        },
        combine = function() paste(fee,fi)
    )
)
Bar  <- setRefClass(
    'Bar',
    contains = 'Foo',
    fields = c('baz','buz'),
    methods = list(
        initialize = function(baz='baz',buz='buz',...){
            baz <<- baz
            buz <<- buz
            callSuper(...)
        },
        concat = function() paste(combine(),baz,buz)
    )
)
```

```
> x <- Bar$new()
> x$concat()
[1] "fee fi baz buz"
> y <- x
> y$fee <- 'yeah!'
> x$concat()
[1] "yeah! fi baz buz"
```

Hypertext Transfer Protocol (HTTP)

- A Client/Server plaint-text protocol. It's dead simple ;)
- Browsers send Request messages, Servers send Response messages.
- Nearly Identical and contain only three (or four if you really count the blank line) parts.
- Request/Response line
- Headers as 'key: value'
- Optional body
- http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Browser requests http://app.rapache.net/helloworld

```
GET /helloworld HTTP/1.1
Host: app.rapache.net
Connection: keep-alive
Cache-Control: max-age=0
Pragma: no-cache
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/
png,*/*;q=0.5
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) AppleWebKit/534.16 (KHTML, like G
ecko) Chrome/10.0.642.2 Safari/534.16
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Server responds with...

```
HTTP/1.1 200 OK
Date: Wed, 09 Mar 2011 17:51:20 GMT
Server: Apache/2.2.12 (Ubuntu)
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

115
<h1>Hello World </h1>What is your name?<form method="GET" action="http://app.rapache.
net/helloworld"><input type="text" name="friend"></form><br>Youre browser type is Moz
illa/5.0 (X11; U; Linux i686; en-US) AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10
.0.642.2 Safari/534.16
0
```

HyperText Markup Language

```
<h1>Hello World!</h1>
<p> This is a summary of Fisher's Iris Data set</p>
<pre>
 Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
</pre>
<p> And here's a plot of the data</p>
<img src="iris.png">
```

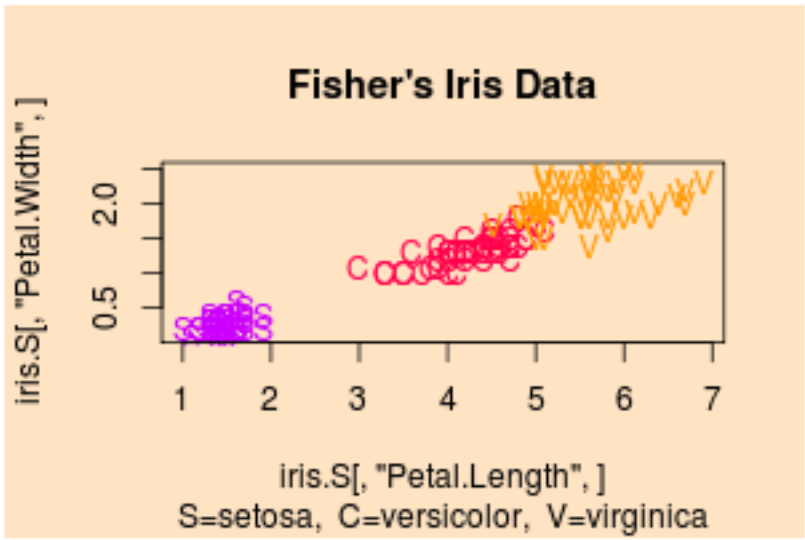
HyperText Markup Language

Hello World!

This is a summary of Fisher's Iris Data set

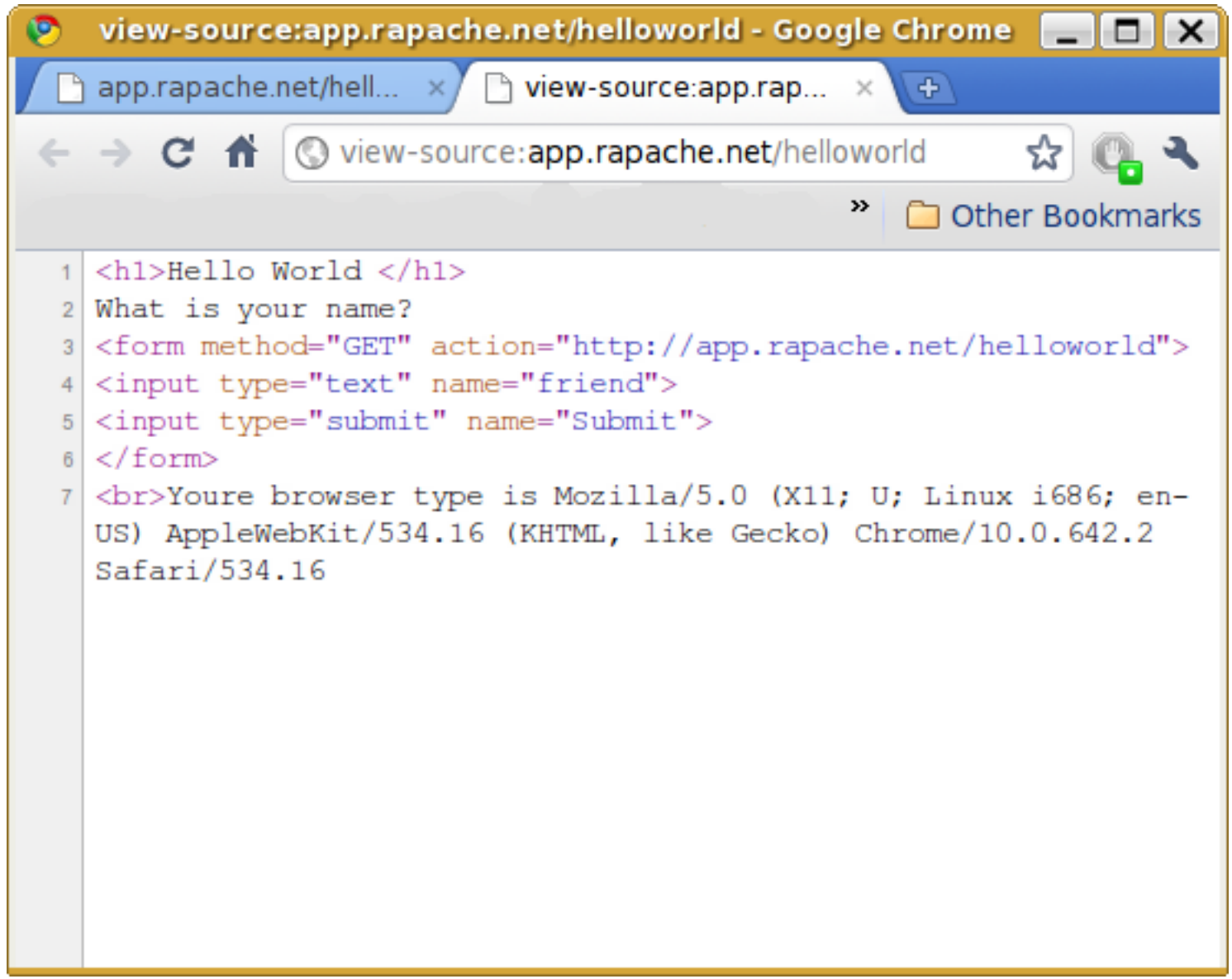
```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

And here's a plot of the data



HTML Forms, HTTP GET & POST

HTML Forms Capture User Input



User clicks on "Submit" button and browser sends this request

```
GET /helloworld?friend=Jeff&Submit=Submit HTTP/1.1
Host: app.rapache.net
Connection: keep-alive
Referer: http://app.rapache.net/helloworld
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,
text/plain;q=0.8,image/png,*/*;q=0.5
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US)
AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.642.2
Safari/534.16
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

- User data now captured as a "query string" encoded with ampersand and equal sign.
- http://en.wikipedia.org/wiki/Query_string

Difference in a GET and POST request

```
POST /helloworld HTTP/1.1
Host: app.rapache.net
Connection: keep-alive
Referer: http://app.rapache.net/helloworld
Content-Length: 28
Cache-Control: max-age=0
Origin: http://app.rapache.net
Content-Type: application/x-www-form-urlencoded

friend=Jeffrey&Submit=Submit
```

Presumes that *helloworld* had written the following form.

```
<form method="POST" enctype="application/x-www-form-urlencoded"
action="http://app.rapache.net/helloworld">
<input type="text" name="friend">
<input type="submit" name="Submit">
</form>
```

User input encodings

- application/x-www-form-urlencoded
  - Can be used in both GET and POST requests
  - http://en.wikipedia.org/wiki/Percent-encoding
- multipart/form-data
  - Used only in POST requests. Useful for uploading files.
  - http://en.wikipedia.org/wiki/MIME

Getting Started

```
> library(Rook)
> s <- Rhttpd$new()
> p <- system.file('exampleApps',package='Rook')
> s$launch(name='RookTest',app=file.path(p,'RookTestApp.R'))
> s$launch(name='hello',app=file.path(p,'helloworld.R'))
> s$launch(name='helloref',app=file.path(p,'helloworldref.R'))
> s$launch(name='hmisc',app=file.path(p,'Hmisc/config.R'))
> s$launch(name='summ',app=file.path(p,'summary.R'))
> s
Server started on 127.0.0.1:35201
[1] RookTest http://127.0.0.1:35201/custom/RookTest
[2] hello http://127.0.0.1:35201/custom/hello
[3] helloref http://127.0.0.1:35201/custom/helloref
[4] hmisc http://127.0.0.1:35201/custom/hmisc
[5] summ http://127.0.0.1:35201/custom/summ

Call browse() with an index number or name to run an application.
> s$browse(1)
```

Rook Iterative Programming Process

1. Load Rook
   - `library(Rook)`
2. Create an Rhttpd variable
   - `s <- Rhttpd$new()`
3. Create your application in a separate file, assign it to either the name of your application or simply just 'app'.
4. Add your application to the server:
   - `s$add(name='MyApp',app='path/to/your/app.R')`
5. Load the app into your browser
   - `s$browse('MyApp')`

If you have errors, **Edit** your app file, **Save**, and **Refresh** browser window. **Repeat until satisfied**!

Examples

- Using Rook's Request and Response classes
- Inputs and Outputs
- Dynamic Graphics

What's in the env?

```
List of 17
 $ HTTP_CONNECTION     : chr "keep-alive"
 $ HTTP_ACCEPT         : chr "application/xml,application/xhtml+xml,text/html;
q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5"
 $ HTTP_ACCEPT_LANGUAGE: chr "en-US,en;q=0.8"
 $ QUERY_STRING        : chr ""
 $ SERVER_NAME         : chr "127.0.0.1"
 $ SCRIPT_NAME         : chr "/custom/RackTest/"
 $ SERVER_PORT         : chr "25434"
 $ HTTP_ACCEPT_CHARSET : chr "ISO-8859-1,utf-8;q=0.7,*;q=0.3"
 $ PATH_INFO           : chr ""
 $ rack.errors         :Formal class 'RhttpdErrorStream' [package "Rack"] with
1 slots
  .. ..@ .xData:
 $ rack.input          :Formal class 'RhttpdInputStream' [package "Rack"] with
1 slots
  .. ..@ .xData:
 $ HTTP_ACCEPT_ENCODING: chr "gzip,deflate,sdch"
 $ REQUEST_METHOD      : chr "GET"
 $ HTTP_USER_AGENT     : chr "Mozilla/5.0 (X11; U; Linux i686; en-US)
AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.642.2 Safari/534.16"
 $ HTTP_HOST           : chr "127.0.0.1:25434"
 $ rack.url_scheme     : chr "http"
 $ rack.version        : chr "1.0"
```