```
browser -> request -> server -> page lookup -> return page to browser -> display
XXX -- REPLACE ME -- XXX
```

Figure 0.1: Basic flow of a how a static HTML file is displayed on a browser.

The internet allows one to distribute their work in a convenient, standardized way that allows people from around the globe to share. Indeed, the R project would never have reached the point it is had it started 10 years prior, as the web technologies that enabled user participation from disparate points did not exist in widespread form.

This chapter shows some of the means to produce interactive interfaces between the user and R through web technologies, at the time of writing. Web interfaces to some process have many obvious advantages over the desktop interfaces discussed in previous chapters: no installation issues for R and the toolkit libraries, user familiarity with a browser, operating system independence, etc. This makes it much easier to share ones work, but also puts an added burden on the GUI writer, who must have some familiarity with new technologies and the security implications contained therein.

This chapter does not even pretend to be comprehensive. It covers an enormous array of technologies. Rather, its focus is to show how R can be used with these technologies. The interested reader will likely need to seek additional help before implementation.

The web programmer coming to R will find relatively simple tools as compared say to some open-source tools available for the python programmer (Django `djangoproject.com`, pyjamas `pyjs.org`, ...) or the ruby programmer (Ruby on Rails `rubyonrails.org`) or even the web programmer used to one of the many available frameworks built on PHP (Drupal `drupal.org`, Joomla! `joomla.org`, ...). However, we will see that there are useful tools for R that make it possible to develop R-driven websites. Of course, web technologies are changing quite rapidly, and R package writers are hard at work, so one should check to see if newer, more powerful resources, have been added to the mix.

0.1 Authoring Web Pages

The simplest web page is a static page that is returned when a user make a request. The basic architecture involves a browser (or some other client) requesting a document from a web server. The request must endcode what document is desired, so the web server can find it. The request is specified in terms of a *URI*, or uniform resource identifier (a URL is technically a type of URI). The web server in turn maps the URI request to a file on the file system which the web server returns to the browser.

The type of HTML file just described is known as a static file, in contrast to a dynamically generated file, as its contents do not reflect any possible extra information in the request. The authoring of static HTML files may involve three different techonologies described next.

Markup languages

Typically a static web page is marked up in HTML. This now familiar markup language allows the page author to indicate structure in various parts of the document. Typical structures are paragraphs, headers, images, etc. Additionally, tags can denote presentation, such as color, font etc.

HTML is centered around the concept of a tag which is used to wrap a portion of the text of a file. A tag has a name or keyword, in lower case, and is enclosed in angle brackets. If the tag encloses some text, it has a start and end style. The start tag for a tag `x` would be `<x>` where the end tag would be `</x>` (an extra slash). All text between theses tags would carry this tag. Some tags, such as the image tag `img`, are used to define their attributes only (a link in this case) so do not come in pairs, in this case it is common practice to end the tag with `/>`. [1]

A few typical tags are specified in Table **??**. Tags may indicate how text is supposed to be formatted (e.g. `b`), others indicate what type of text it is (e.g. `code`), others the document structure (e.g., `h1`, `p`, etc.).

A tag may have one or more *attributes* specified. For example, an anchor tag, `a`, has an attribute to specify the image by URI, `href`. This attribute is indicated by name with an equals sign. Quotes are optional for HTML, but recommended in general. They are mandatory if there is white space involved. An example might be `<a href="http://www.r-project.org" />`.

All tags have an `id` attribute, which is used to give a unique ID to the part enclosed by the tag. This is used to identify the tag within the document object model (DOM) described in brief later. All tags also have a `class` attribute to indicate if the tagged content should be treated as a member of a class. This provides a means to classify and treat similar objects as a group. Some tags also allow one to specify style attributes, but a more modern approach is to use a stylesheet to specify those. The `span` and `div` tags are primarily used to specify attributes for the tagged text.

Some characters, such as angle brackets, have a reserved meaning. As such, to use an angle bracket in an HTML document requires the use an *HTML entity reference*. There are many such entities – they are also used for

[1] There are two common variants of HTML one coming from SGML, another, XHTML, deriving from XML. Both are similar, but xhtml is stricter with its use of tags. Some basic rules (as opposed to conventions) include all tags are either ended with a closing tag, or with `/>`; tags are lower case; attributes must be enclosed in quotes and specified; the root element is different from that given in the examples. The Web Hypertext Application Technology Working Group (`http://whatwg.org`) has proposed specifications for the two that seem likely to become the standard for HTML5.

Table 0.1: Table of common tags in HTML.

Tag	Description
html	Denotes an HTML file
head	Marks header of file
body	Marks off main body of file
script	Used to include other types of files
p	A paragraph. Also, br for a line break
h1	First level header. Also h2,...,h6
ul	Unordered list. Also ol.
li	Denotes a list item
a	An anchor for a hyperreference
img	Denotes an image
div	A text division, indicates a line break
span	A text division, no implied break
b	Denotes text to be set in bold
code	Denotes text that is code
em	Denotes text to be emphasized
table	Creates a table element

cahracter encodeings. Entities are denoted by a leading ampersand `&` and trailing semicolon, as with `<`.

Example 0.2: Simple HTML file
A basic HTML file would include a structure similar to the following which shows the head and body. Within the head, a title is set.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>Hello World Page</title>
  </head>
  <body>
    Hello world
  </body>
</html>
```

A basic `xhtml` file has a different header, for example the following which specifies a version for the XML and a default name space through the `xmlns` attribute.

```
<?xml version="1.0" ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<head>
<meta http-equiv="content-type"
```

```
  content="text/html; charset=UTF-8"/>
<title>Page title</title>
</head>
```

Example 0.3: A basic table
Displaying tables is a common task for web pages. The `table` tag encloses a table. New rows are enclosed in a `tr` tag, and each cell can be a header cell, `th`, or a data cell `td`. The following shows one way alternate rows can be striped by hard coding a color attribute to the rows.

```
<table border="0" cellpadding="0">
  <tr>
    <th>Header 1</th><th>Header 2</th>
  </tr>
  <tr>
    <th>1</th> <th>2</th>
  </tr>
  <tr bgcolor="goldenrod">
    <th>3</th> <th>4</th>
  </tr>
</table>
```

Example 0.4: R helpers
Writing a tag and specifying a table can be tedious, especially if the data is in an R object. Some helper functions are useful. The `hwriter` package includes a few, for now we mention `hmakeTag` which will produce a tag around some specified content along with attributes, that can be passed in through R's `name=value` syntax.

```
require(hwriter)
out <- hmakeTag("td",1:2, bgcolor="red")
cat(out, sep="\n")
```

```
<td bgcolor="red">1</td>
<td bgcolor="red">2</td>
```

The function is vectorized, as can be gathered from the output.

Style sheets

Casscading Style Sheets (*CSS*) may be used to specify the presentation of the text on a page. Common practice is to use the markup language to specify document structure and a separate document to specify the layout of the first document. The advantage is a clean separation of tasks so that one can make changes to the layout, say, without affecting the text (and vice versa). A typical usage is to be able to provide different layouts depending on the type of device.

Without going into detail, the style sheet syntax provides a means to specify what type of tagged content the style will apply to (the selector) and a means to specify what styles of markup will be applied. For example, the specification below has `h1, h2, h3, h4, h5, h6` as a selector to indicate that it applies to all header tags. In the *declaration block* are style specifications for the color of the text and the font weight. Additionally, specifications for margins and padding are given, along with a border on the bottom around the element.

```
  h1, h2, h3, h4, h5, h6 {
    color: Black;
    font-weight: normal;
    margin: 0;
    padding-top: 0.5em;
    border-bottom: 1px solid #aaaaaa;
}
```

The full specification allows for much more complicated selections, be they based on id of the tag, class of the tag, or relation of tag to an enclosing tag. Style sheets can also refer to positioning of the object within the page. Most modern web pages use style sheets for layout, rather than tables, as it allows for greater accessibility and offers advantages with search engines.

Example 0.5: A striped table using style sheets
Using the `bgcolor` attribute of a table is deprecated in favor of style sheets for good reason. Here we illustrate a style sheet approach to striping a table. The style sheet may be defined in the HTML file itself with the `style` tag that appears within the document's `head`.

```
<style type="text/css">
table { border: 1px solid #8897be; border-spacing: 0px}
tr.head {background-color:#ababab;}
tr.even { background-color: #eeeeee;}
tr.odd { background-color: #ffffff;}
</style>
```

A more common alternative, is to use the `link` tag to include the stylesheet through a url. For example,

```
<link rel="stylesheet" href="the.url.of.the.sheet" />
```

For the table itself, we need only replace the specification of the attribute with a class specification.

```
<table>
  <tr class='head'>
    <th>Header 1</th><th>Header 2</th>
  </tr>
  <tr class='odd'>
```

```
    <th>1</th> <th>2</th>
  </tr>
  <tr class='even'>
    <th>3</th> <th>4</th>
  </tr>
</table>
```

JavaScript

The third primary component of most modern web pages is JavaScript. This is a scripting language that runs within the browser that allows manipulation of the document. The document object model (DOM) specifies the elements of the text that may be referenced from within JavaScript. For example, individual elements can be found by unique ID, or common elements by class, or elements sharing a tag, say `p`. JavaScript provides methods for manipulating these elements. The simplest uses might be to change the text when the mouse hovers over an element.

JavaScript allows web pages to be dynamic interfaces. The language allows for callbacks to be defined for certain events, as with the other GUI toolkits we've discussed. We don't pursue this, but note that the `gWidgetsWWW` package uses JavaScript to make dynamic web pages (cf. Figure **??**).

Example 0.6: Simple use of JavaScript to make a button have an action
The `button` tag produces a visual button. This tag has several event attributes, including `onmouseover` and `onclick`. Here we show how to change the documents background color on a mouseover, and how to display a message on a mouse click.

```
<button
  onMouseOver="document.bgColor='red'; return true;"
  onMouseOut="document.bgColor=''; return true;"
  onClick="alert('clicked button'); return true;" >
  Click me...
</button>
```

There are several open source JavaScript libraries available that offer convenient interfaces to JavaScript and UI widgets. We mention ExtJS (`www.extjs.com`, jQuery (`jQuery.com`), YUI (`developer.yahoo.com/yoi`) and Dojo (`www.dojotoolkit.org`).

R tools to assist with authoring web pages

There are quite a few packages for R to faciliate authoring web pages from R objects. We mention a couple.

The hwriter package

The `hwriter` package simplifies the task of creating HTML tables for R objects, such as a matrix or vector. The package has a self-generated example page in HTML which is created by its `showExample` function. The main function `hwrite` maps R objects into table objects (by default) and has many options to modify the attributes involved. By default, it writes its output to a file. The helper function `openPage` takes a file name and returns a text connection. The `closePage` function will close it. In the examples below, so as the output will print, we use the `stdout` function instead for the connection.

The package's examples show many different usages, we illustrate a few below.

To create a simple table, we simply call the constructor on a matrix or data.frame object:

```
m <- matrix(1:4, ncol=2)
hwrite(m, page=stdout())
```

```
<table border="1">
<tr>
<td>1</td><td>3</td></tr>
<tr>
<td>2</td><td>4</td></tr>
</table>
```

To get alternate rows to be striped we could have:

```
hwrite(m, page=stdout(),
       row.class=rep(c("odd","even"), nrow(m))[1:nrow(m)])
```

```
<table border="1">
<tr>
<td class="odd">1</td><td class="odd">3</td></tr>
<tr>
<td class="even">2</td><td class="even">4</td></tr>
</table>
```

(Or using the `bgcolor` attribute setting `row.bgColor`. Although, most of the `hwrite` arguments recycle, in this case we do so manually. as the "`row.`" attributes need such assistance.)

Finally, a hyperlink can be generated through the `link` argument.

```
hwrite("R project", link="http://www.r-project.org",
       page=stdout())
```

```
<a href="http://www.r-project.org">R project</a>
```

The R2HTML package

The R2HTML provides the generic function HTML for creating HTML output from R objects based on their class. As with hwrite, this function writes its output to a connection for ease of generating a file.

As HTML is a generic function, its usage is straightforward. For a numeric vector we have:

```
library(R2HTML)
HTML(1:4, file=stdout())
```

```
<p class='integer'>1  2  3  4</p>
```

The class is written using the class attribute, so a style sheet can be used:

```
HTML(c(TRUE, FALSE), file=stdout())
```

```
<p class='logical'>TRUE  FALSE</p>
```

Functions may be formatted:

```
HTML(mean, file=stdout())
```

```
<br><xmp class=function>function (x, ...)
UseMethod("mean")
<environment: namespace:base></xmp><br>
```

For more complicated objects, such as matrices and data frames, the HTML function has other arguments. For example, a border and inner border can be set (we omit the output).

```
HTML(iris[1:3,1:2], Border=10, innerBorder=5, file=stdout())
```

The package also includes a number of functions to facilitate the drafting of HTML files within R, including HTMLInitFile, HTMLCSS, HTMLInsertGraph and HTMLEndFile. The functions HTMLStart and HTMLEnd

The brew package

R has the wonderful facility Sweave that passes through a LaTeX file and can replace R code with the code and output generated by evaluating the code. The R2HTML provides a means to do the same with HTML files. Whereas, the ascii package provides a means to do so for several ascii-based syntaxes for markup, many of which have tools to create HTML pages.

The brew package does something similar, yet different. It allows one to place a template within an HTML file that R will eventually populate when called accordingly. In the next section, we illustrate how this can be used dynamically in a web page. For now, we mention how to make a template and how to process it.

		Evaluate	
		Yes	No
Print	Yes	<%= %>	no delimiters
	No	<% %>	<%# %>

Table 0.2: The `brew` delimiters and how they are processed

A template is a file which is processed by the `brew` function. This function returns the contents of the file after modification to store in a file or to print out.

A template is a file with parts of it marked by delimiters (cf. Table **??**). All text not within delimiters is processed as is. Whereas, text within delimiters may be evaluated by R, and if evaluated the contents may be inserted into the output or simply used to adjust the evaluation environment.

Example 0.7: Differences in `brew` delimiters

To illustrate the differences in the `brew` delimiters, the left side has `brew` commands and the right side is their output.

```
Run, no print <% x <- 4 %>
Eval and print <%= x^2 %>
Comment <%# A comment %>
Inline <%= x -%>  value
```

```
Run, no print
Eval and print 16
Comment
Inline 4  value
```

The inline example has a dash before the closing delimiter to suppress a newline.

Example 0.8: Dynamically formatted text
This example shows how brew can be used to insert dynamic text.

This template

```
<%
  require(fortunes)
  out <- fortune(155)
%>
<h3 class="fortune">Fortune</h3>
<%= wrap(out$quote) %> <br>
--<em><%= out$author %></em>, <%= out$date %>
```

produces

```

<h3 class="fortune">Fortune</h3>
It might surprise many R-help posters, but R has
manuals as well... <br>
--<em>Uwe Ligges</em>, January 2006
```

Example 0.9: Recursively calling `brew`
Typically there will be more than one page on a web site with each sharing common features: a banner, a footer, navigation links, a side bar, ... Using templates for these pieces and then including the template in a file is one way to centralize these common pieces. The `brew` function can easily be used to do this.

For example, here we define a header and footer and then call them in from a page. Our header is basic template, but includes a variable `title` to be defined in the page.

```
<html>
<title><%= title %></title>
</html>
<body>
```

Our basic footer is

```
<div id="footer">
     [boilerplate text goes here]
</div>
</body>
```

And a typical page has this structure. We set the variable `title` in the scope of this page, but it is seen withing the scope of the call to process the header page.

```
<% title <- "A sample page" %>
<%= brew("brew-header.brew") %>

A basic page

<%= brew("brew-footer.brew") %>
```

Example 0.10: Creating a template within a template
This example shows how one can define a template within a template, as an alternative to a separate file. The basic idea is to use `paste` to bypass the issue of being unable to nest `brew` delimiters. We evaluate the template within a context, so that each time we get the values from different rows.

This template

```
<% tmpl <- paste("<a href=<",
   "%= URL %", ">><",
   "%= Name %","></a><br />",
   sep="")
%>
<%
df <- getCRANmirrors() ## some data frame
for(i in 2:3) {
```

```
   context <- df[i,]
   with(context,  brew(text=tmpl))
}
%>
```

produces

```
<a href=http://cran.ms.unimelb.edu.au/>Australia</a><br />
<a href=http://cran.at.r-project.org/>Austria</a><br />
```

Graphics in web pages

Web pages may be plain text, but most contain images or graphics. The `img` tag allows one to display a graphics file in an HTML page by specifying it `src` attribute. This is an image file, often in `png`, `gif` or `jpeg` format. In this section, we describe how R can be used to generate images by using different device drivers. To list all possible stock devices, see the help page for `Devices`. The function `capabilities` list which devices are available for a given R installation.

png

Typically when a plot command is issue, an interactive plot device is opened or reused, however, the user can specify a device to save the output to a file for further use. For example, the `pdf` and `postscript` functions will turn R commands into files for inclusion in written documents. For web pages, the `png` and `jpeg` device drivers are available for many systems. These may be used to insert a graphic into a web page.

The basic usage is like that of the `pdf` driver illustrated below – open the device, issue graphics commands, close the device:

```
pdf(file="test.pdf", width=6, height=6) # in inches
hist(rnorm(100), main="Some graphic")
invisible(dev.off())                    # close device
```

To use the `png` driver on a linux server, the option `type` should be set to `cairo` either through the constructor, or by setting the option `bitmapType`.

The `Cairo` device driver is an alternative which can also output in png format.

SVG graphics

The web has other means to display graphics than an inclusion of an image file. For example, Flash is a very popular method. [2] Scalable vector graphics (SVG, `http://www.w3.org/Graphics/SVG/`) is another way to specify a

[2] The `FlashMXML` from `omegahat.org` provides a means to genearate flash files from within R.

graphical object using XML. Many modern web browsers have support for the display of SVG graphics. To insert the file, we have the `object` tag and its attributes `data` and `type`, as in

```
<object data="image-svg.svg" type="image/svg+xml"></object>
```

Not all browsers support svg, so one might also have a fall back image, as in:

```
<object data="image-svg.svg" type="image/svg+xml">
<img src="image-png.png" alt="alternative file" />
</object>
```

There are a few drivers to create SVG files in R, for example In the base `grGraphics` package, the driver `svg` is available. This non-interactive driver is used as the `png` one illustrated above.

The `RSVGTipsDevice` package provides an alternate driver, `devSVGTips`. The "Tips" part of the package, is provided by the function `setSVGShapeToolTip`, which allows one to specify a tooltip to popup when the mouse hovers over an element. The tooltip specified is placed over the next shape drawn, such as a point.

For example, here we add a tip and a URL to each point in a scatterplot. We initially call `plot` without plot characters to set up the axes, etc.

```
require(RSVGTipsDevice)
f <- "image-svg.sv"
devSVGTips(f, toolTipMode=2, toolTipOpacity=.8)
plot(mpg ~ wt, mtcars, pch=NA)
nms <- rownames(mtcars)
for(i in 1:nrow(mtcars)) {
  ## need to add tooltip shape by shape
  setSVGShapeToolTip(title=nms[i])    # add tooltip
  setSVGShapeURL("http://www.google.com")    # some URL
  with(mtcars, points(wt[i], mpg[i], cex=2, pch=16)) # add
}
invisible(dev.off())
```

The canvas tag

HTML5 is a major extenstion to HTML that is being implemented in most browsers at the time of the writing of this book. One of the new features of HTML5 is the `canvas` element, which allows JavaScript code to manipulate objects, similar to the `tkcanvas` widget of `tcltk`.

R has the `canvas` device driver, that can be used to generate JavaScript code to produce the graphic in a canvas element. The basic usage involves creating the JavaScript:

```
require(canvas)
```

```
f <- "canvas-commands.js"
canvas(width=480, height=480, file=f)
hist(rnorm(100), main="Some graphic")
invisible(dev.off())
```

Then, within the HTML File, code along the lines of the following is needed. The first `script` tag is used to define the variable `ctx` to hold the canvas object, as this is assumed by the canvas package.

```
<canvas id="canvas_id" width=480 height=480></canvas>
<script type="text/javascript" language="javascript">
var ctx = document.getElementById("canvas_id").getContext("2d");
</script>
<script type="text/javascript" src="canvas-commands.js"></script>
```

An alternative to `canvas` is possible through the `RGraphicsDevice` device from `omegahat.org`.

0.2 The rapache package

While websites can consist of just static files, many webpages viewed are dynamically generated in response to user input. In order to implement this, the process of returning a page for a user request is more complicated. Rather than simply look up a file, the web server may call an external program that prepares the text to return. This text may be HTML for a web page, or in the case of web services, may be XML or some other form of data markup. For R users, there have been a few projects in the past that allow an R process to be used to generate the response. At this point, the best one is the `rapache` package. The package web page lists a few projects that use this technology to create web pages, including some highly interactive web pages by Jeroen Ooms. The `gWidgetsWWW` package ports the `gWidgets` API to the web using `rapache`.

The *Apache* web server is one of several open-source projects supported by the apache Apache Software Foundation. It is extremely successful – its website (`http://www.apache.org`) boasts it has been the most popular web server on the internet since 1996. Like R, Apache's open source nature allows developers to customize its standard behavious, in this case using modules. The `RApache` package (`http://biostat.mc.vanderbilt.edu/rapache/`) provides such a module that inserts R in the processing phase of a request to the web server.

The `rapache` package works under linux but not directly under windows. However, one can use a virtual machine to run a linux version of Apache under windows or Mac OS X. A "virtual machine" containing a pre-built linux system is available from the `rapache` website.

```
request url -> mangle file name -> lookup, return file

 to

request url -> mangle file name -> run function file through rapache handler
            -> return output
```

Figure 0.2: Inserting `rapache` in the request lookup

Configuration

The `rapache` package requires the Apache web server to be properly configured. There are a number of steps in the process. The `rapache` homepage has detailed instructions, we mention just the steps here.

First, a module for Apache must be created by running `rapache`'s configure script. For Debian users, the package can be installed through the usual mechanism. Afterwards, Apache must be configured.

The R module must be loaded into Apache. This is done in the standard way for Apache, through its `LoadModule` directive. This is done before any other R-centric directives are given in Apache's configuration.

The `REvalOnStartup` directive is used to specify any packages that should be loaded whenever the web server starts. The web server embeds a copy of R in itself and spawns copies of this as it spawn copies of itself to handle requests. The startup can be slow, so this offers a chance to pre-load common packages to speed things up at the cost of a larger memory footprint. `RSourceOnStartup` is similar, only it used to specify a file to be sourced on startup.

The Directory Directive There are a few directives to configure `rapache` to process an incoming request. A standard configuration for Apache, is to have the URL specify a file on the file system after some mangling of the name, exchanging the base part of the URL with a document root. One can have `rapache` process the file prior to being returned by creating the appropriate directive

The `rapache` manual specifies a typical usage is to call `brew` on the file. That is, to make a dynamic web page a brew template is created and placed into the appropriate directory.

To configure `rapache` to do this, the following example directive may be added to Apache's configuration.

```
<Directory /var/www/brew>
  SetHandler r-script
  RHandler brew::brew
</Directory>
```

```
request -> rapache calls function -> returns output to client
```

Figure 0.3: Creating a web page from a script and inputs

If the "DocumentRoot" of Apache is `/var/www`, then a request such as `http://servername/brew/file.brew` will resolve first to Apache finding `file.brew` in the `/var/www/brew/` directory, and then that file will be processed by the `brew` function in the `brew` package. The output will then be returned to the client making the request. The `SetHandler` directive, specifies the handler will be script, thereby passing along the file information and an environment. The `RHandler` directive is used to specify a function to call. This example assumes the `brew` package is not loaded, so the `brew` is found from within the package. An alternative to `brew`, may be `sys.source`.

The Location Directive Requests need not map to a file system, but can simply map to a function call. For example, an application might be designed around data stored in a data base and all pages are generated dynamically. To have a URL call a script without reference to a file, the `LOCATION` directive is used. For example,

```
<Location /myapp>
  SetHandler r-handler
  RFileHandler /path/to/R/scripts/myapp.R
</Location>
```

A request to `http://servername/myapp/extra` will call the script `myapp.R`. The `extra` part of the request can be found from one of the `rapache` variables discussed in Section **??** and the script can adjust its output based on this.

Creating files

The typical use of `rapache` is to return an HTML file, but this is not the only possibility.

A `brew` template can easily have the necessary header/body structure of an HTML file to be returned by the server to the client. However, if the page is generated by a function call, as with the Location directive, `rapache` provides some convenience functions.

Response headers can be added throught the `setHeader` function. The set of headers is long and technical. [3]

Web servers typically return HTML files, but are possible of much more. For example, the server may be asked to dynamically generate a graphic,

[3]The definitions can be found at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`.

and the output would be an image file. As well, web services are used to pass some resource, say some data to a client requesting it. This data may be stored in XML format, or JSON or YAML etc. The `setContentType` function is used to set the MIME type of the response. It must be called before any `print` or `cat` statements in the file. To send back binary data, the function `sendBin` is available.

Return Codes The return value of the handler call indicates the failure or success of the request. The return value should be an integer, `rapache` provides named variables instead. For success a return value of `DONE` will indicate success, whereas a value such as `HTTP_BAD_REQUEST` will signal an error. [4].

The function `RApacheOutputErrors` can be used to direct what happens to the error, in particular it can be used to have errors print out to the browser rather than the log file. This is useful when developing a program.

`rapache` variables

When a script of function is being evaluated within `rapache` certain variables holding information about the request and web server are created. The variables are lists with named arguments, the names matching `Apache` variables.

`SERVER` The `SERVER` variable holds a large amount of information on the request. For example, the `status` componenent returns the status code. Some of the most useful, decompose the URL requesting the page.

The response depends on the configuration. If the we use `/var/www/brew` to process requests through `brew`, as above, then a request like `http://localhost/brew/test.brew?some=brew` results in values of `uri` being `/brew/test.brew`, `filename` being `/var/www/brew/test.brew`, `path_info` being an empty string and `args` holding the string `some=brew`.

However, if we use the Location directive above, then the request `http://localhost/myapp/detail?some=brew` has `uri` being `/myapp/detail`, `path_info` being `/detail` (the "virtual" part of the request), and `args` again holding the string `some=brew`.

`GET` Both of the example urls above result have `SERVER$method` being `GET`. HTTP has a few conventions that are not enforced, but are associated with it providing RESTful web services. One being that one uses a certain set of methods to interact with the service. A `GET` request is meant to return data, a `POST` request is meant to create new data, a `PUT` request is meant to update data and a `DELETE` request to delete data.

[4] A list of the `rapache` variables appear in its manual. A list of status codes can be found at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

The two example requests above, result in GET reqests and the GET variable contains some useful information, namely the arguments passed through the URL after the ?. (URLs use a ? to pass arguments in the form key=value with multiple arguments separated by an &. So in the above, GET is a list with component some whose value is brew.

POST A POST request usually comes from within a form. As with a GET request, arguments can be passed in with the request, although they do not appear in a URL as illustrated above. As with the GET variable, the arguments appear as named components in the POST variable. POST request can contain more information – they are not limited in length the same way – and must be used to upload files, say.

COOKIES By design HTTP is a stateless protocol. This means that between requests the web server remembers nothing about the past requests. For large web sites, this has an advantage when multiple servers are used to process requests. However, it has disadvantages as the request must relay the state of a web page. Several mechanisms have been developed to deal with this issue. Sessions, where information is kept server side and an ID kept with the client allows a state to be maintained. Another solution is to store information on the client side. This is implemented through cookies. Although cookies have privacy issues, their use is widespread.

A basic cookie consists of a name and a value (a character vector of length 1). Cookies must satisfy certain validity constraints which are specified through a time to expire, a path to which the cookie pertains and a domain for which the cookie is valid. The rapache function setCookie can be used to set a cookie. The first argument is the cookie name, the second the value, and others are available to set properties, such as an expiry time. Cookies are placed in the outgoing header of a document, so this call is done before the head tag.

When a page is loaded, the COOKIES variable contains cookie information. Again, as a list. In this case, the names are the valid cookie names and the component's value is the cookie.

Forms

User input can be passed to the server through the URL request or through a form. Forms are specified with the form tag, which has a few important attributes. [5]. The action attribute specifies the URI that will process the form information. In our example, this will match a Location directive. The method attribute is used to specify a GET request or a POST request. For a post request that includes a file upload, the enctype attribute should contain

[5] See http://www.w3.org/TR/html401/interact/forms.html for a specification

"multipart/form-data". In addition to these, the onsubmit attribute is often used to specify some JavaScript to call as the form is submitted. For example, this may be used to specify code to validate the form entries.

The input tag With the form tags control elements may be placed. The input tag is used to specify several types of controls. The type attribute is used to specify which control. The default is text for a single line text entry, but other values are password for a password entry; checkbox and radio for selection of items; file for a file upload control; image for an image; button to make a button; and submit for a submit button.

The usual attributes class and id apply, as do many others that are type specific. The name attribute specifes the name for the element. This is processed as a key in the POST variable. The value attribute is used to specify an initial value. For sizing, the attributes size and maxlength are used to specify the control size and length of text string. For images src is used to specify the image source as a URL. For the selection widgets, checked is used to specify if the button is on.

To illustrate, this HTML would produce a simple text entry area

```
<input type="text" value="initial text" />
```

This would be used to specify a submit button

```
<input type="submit" value="submit" />
```

A radio group is created by having multiple inputs sharing a common name

```
<input type="radio" name="key" value="TRUE" checked="TRUE">
<input type="radio" name="key" value="FALSE">
```

The select tag The select tag is used to create a control to select one or more values from a list of options. This control may be a combobox or a table display. The attribute multiple is used to specify if the user can select one or more values. When specified, the POST or GET variables have multiple components of the same name. The size attribute specifes the number of entries to make initially visible.

The possible values for selection are given within option tags. The attribute selected is used to specify if the value is initially selected. The attribute value can be used to specify a different value than that displayed.

For example,

```
<select name="id">
  <option value="1">one</option>
  <option value="2" selected="true">two</option>
</select>
```

A textarea tag Single line text entries are created by the `input` tag by default, but multiple line entries are formed by the `textarea` tag. The attributes `cols` and `rows` specify the size.

Security

Forms allow users to specify values. These value may then be processed by the underlying R process within `rapache`. As such a malicious user may try to have code run that could compromise the web server. More benignly, the user may specify responses that include malformed HTML. If these are simply printed back when the web page is created, a rendering error may occur. Regardless of the user base for a web application, one should assume that user input for web sites should never be trusted.

Unclosed or malicious tags To avoid malformed HTML one should encode any user input that is echoed back to a web page. The following function will replace certain characters with their HTML entity for safe inclusion with a page.

```
HTMLencode <- function(str) {
  str <- as.character(str)
  vals <- list(c('&','&amp;'),
               c('"','&quot;'),
               c('<','&lt;'),
               c('>','&gt;')
               )
  for(i in vals)
    str <- gsub(i[1],i[2],str)
  str
}
```

Whitelists, Blacklists Even in the event of a fixed list of values for a user to choose from, user input should always be checked. It is very easy to fabricate a request without going through the web form, for example the R package `Rcurl` can do this.

When checking values, one can use a whitelist – a list of acceptable values, or a blacklist – a list of unacceptable values. The use of a whitelist is better if possible, as it is very easy to miss somthing in a blacklist.

In either case, it is a good idea to never evaluate directly a users input.

SQL injection Many web sites are built around queries to a data base. Websites powered by `rapache` can take this approach, as the `Rdbi` package allows an interface within the R process between a data base and R. The basic use is to create a query within R and then call one of `Rdbi`'s functions to get the

results from the query. The technique of SQL injection, takes advantage of carelessly constructed SQL queries that are made by pasting together SQL commands with user-given input.

Example 0.11: Using `rapache` to explore a data store

This example shows how one can use `rapache` to allow a user to explore a data set. This basic application is simple, but the structure of it is typical and very extendible. There are three pages to display: a page to greet the user, a page to select one of many items, and a page to display detail on an item.

We use a `Location` directive for this application which allows us to specify which page to display using the `path_info` variable.

```
<Location /simpleapp>
   SetHandler r-handler
   RFileHandler /var/www/GUI/simpleapp/app.R
</Location>
```

The script `app.R` is responsible for processing the request and dispatching to the appropriate page. Our script contains the following to load packages and set the current working directory to match that of the script. This is needed for our calls to `brew`.

```
require(brew, quietly=TRUE)
require(hwriter, quietly=TRUE)
dir <- "/var/www/GUI/simpleapp"
setwd(dir)
```

We have four main pages, one for any errors, and the three mentioned. The dispatch to the page will call these functions which are responsible for setting the context for the `brew` templates. Each template has a `title` variable that we set within the function. This then will be within the scope of the call to `brew`. The variable `df` is assumed to contain a data frame of interest. This could be retrieved by some call to a data base, for example.

Our error page is called by

```
processError <- function(e) {
  title <- "Error"
  with(e, brew("error.brew"))
}
```

The `error.brew` template has

```
<% brew("brew-header.brew") %>
<h2>
  <%= message %>
</h2>
<% brew("brew-footer.brew") %>
```

where the value for message is passed in through the error call. The header and footer templates are straightforward, and are used to give a consistent look to each page. In this case, as we use xhtml, we have for the header:

```
<?xml version="1.0" ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<head>
<meta http-equiv="content-type"
  content="text/html; charset=UTF-8"/>
<title> <%= title %> </title>
</head>
<body>
<%
  if(exists("user_name") && nchar(user_name))
    cat(sprintf("<h2>Welcome %s</h2>", HTMLencode(user_name)))
%>
```

The user_name variable is set in the greeting page, so may not be present. Note the call to HTMLencode to ensure that the value for the name, which comes from the user, does not contain any malformed HTML.

The footer simply closes the body and html tags. In both cases, these templates could be much more complicated.

Our greeting page illustrates how to use a form to gather user input, in this case a name, but in general this might be used for authentification etc.

```
showLogon <- function() {
  title <- "Logon"
  brew("login-form.brew")
}
```

The main part of the brew template is a basic form using the input tag in two different ways.

```
<form method="POST" action="/simpleapp/select">
<label>Enter your name:</label>
<input type="text" name="name" />
<input type="submit" value="submit" />
</form>
```

We use a POST call, as this may be used to modify a data source. As well, the action specification uses select so that the path_info variable can be used to determine which page to call.

After logging on, the user may be asked to narrow the search for data. In this example, the user is asked to select one of the rows of the data source. We generically refer to the row identifier as ID.

```
selectID <- function() {
  title <- "Select an ID"
  context <- list(nms=rownames(df))
```

```
    with(context, brew("select-id.brew"))
  }
```

The `context` variable is used to pass in different contexts to the `brew` template. Of course this could also appear directly in the template, but it is better to separate the logic from the presentation. In this case, the template for ID selection includes this

```
<form method="GET" action="/simpleapp/id">
<select name="id">
<%
  cat(hmakeTag("option", nms))
%>
</select>
<input type="submit" value="submit" />
</form>
```

We use `GET` for the method, as we assume this is merely a request to narrow the display of data, not modify the data store. The useful `hmakeTag` function is employed to vectorize the creation of the HTML `option` tags.

Finally, our call to show detail on the selected identifier includes matching the user specified ID against a list of possible values (a whitelist). If no match occurs, an error message is printed.

```
 showID <- function() {
   title <- "Show an ID"
   id <- GET$id
   if(! id %in% rownames(df)) {
     processError(list(message="id does not match"))
   } else {
     context <- list(d=df[id,], id=id)
     with(context, brew("show-id.brew"))
   }
 }
```

For the display, we have this basic template which uses `hwrite` to put the output into a table.

```
<h3> Detail on <%= id %> </h3>
<%
  hwrite(unlist(d), page=stdout())
%>
```

The main script must figure out the `user_name` variable. This may come from the greeting page through a POST request, or may be stored using a cookie to make the name persistent. This leads to the following (`get_d` is used to provide a default, if the variable is `NULL`):

```
 user_name <- ""
 if (!is.null(POST)) {
```

```
  user_name <- get_d(POST$name, "")
}
if(user_name == "" && !is.null(COOKIES)) {
  user_name <- get_d(COOKIES$name, "")
}
```

Finally, the script is used to dispatch to the proper page. We start by setting the content type and a cookie to store the `user_name` variable.

```
setContentType("text/html")
if(user_name != "")
  setCookie("name",user_name)
```

Following how django processes URLs we set up a list of regular expressions to check against `path_info` and function names to handle the dispatch.

```
urls <- list(select=list(regexp = "^/select", call="selectID"),
             id =   list(regexp = "^/id",     call="showID" )
             )
default_call <- "showLogon"
```

With this, we then process the request as follows.

```
path_info <- SERVER$path_info
flag <- FALSE
for(i in urls) {
  if(!flag && grepl(i$regexp, path_info)) {
    flag <- TRUE
    tryCatch(do.call(i$call, list()), error=processError)
  }
}
if(!flag)
  tryCatch(do.call(default_call, list()), error=processError)
```

We wrap the call inside `tryCatch` in case the page creation throws an error.

The last line of the script is simply `DONE` to indicate to the client that the request is finished.

0.3 Web 2.0

The term web 2.0 is used to describe highly interactive web sites. A key feature of many of these is the use of *Ajax technologies*. The packages `Rpad` and `gWidgetsWWW` use Ajax technologies for interactive web sites. "Ajax" comes from **a**synchronous **J**avascript and **X**ML. The term asynchronous refers to pieces of a web page being updated independently of others, unlike in the previous section where each request creates a new page. The JavaScript term is a substitute for a browser side language to manipulate the web pages

DOM, and XML simply a means to encode data, and shouldn't be taken literally, as other common text-based encodings are used, such as JSON.

Several JavaScript libraries are built around Ajax technologies, such as the `extjs` library and `jQuery`. These provide a means to query a server asychronously through an *XMLHttpRequest*. This section discusses briefly how to use `rapache` to provide the data for such a request.

Example 0.12: Creating a web service using `rapache`
This example will illustrate how make a web service with `rapache`. There are two pieces, the JavaScript code in the web page, and the server code. For the JavaScript piece, we use the `jQuery` library, as the use is somewhat straightforward.

We illustrate how to return HTML, JSON and XML. First, the HTML.

In the header of our web page, we must call in the `jQuery` JavaScript library. These files may be on local webserver, or called in with the following HTML code:

```
<script
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.3/jquery.min.js"
  type="text/javascript">
</script>
```

Inside the HTML page, we have a place holder to put the text from the web service. We use `div` tag, with an unique id.

```
<div id="htmlTarget"> [HTML target] </div>
```

(There are also similar areas for JSON and XML.)

We want the request for data to happen when the page loads. The `jQuery` library provides a means to have a function called as the page loads (before any images are downloaded, say). We place the command within this bit of JavaScript.

```
<script type="text/javascript">
  $(document).ready(function(){
    // JavaScript commands go here
  })
</script>
```

As for the JavaScript commands, the following jQuery code will produce the Ajax request. This assumes the webserver is running locally. One would replace `localhost` with the appropriate site.

```
$.ajax({
   type: "GET",
   url:"http://localhost/ajaxapp/html",
   dataType: "html",
   success: function(data) {
     $("htmlTarget").html(data);
```

```
  },
  error: function(e) {
    $("#htmlTarget").html("<em>Service is unavailable</em>");
  }
  });
```

To explain, the `$` is a `jQuery` variable, the first occurence is a call to its `ajax` method, The arguments are specifed to make a `GET` request to a certain url. The return data will be HTML. The request, if a success, will replace the HTML code within the node with id `htmlTarget` with that returned by the Ajax request. If an error is returned, an error message is placed there instead.

Within the R script run by `rapache`, we have a call like this

```
show_html <- function() {
  require(hwriter, quietly=TRUE)
  setContentType("text/html")
  hwrite(d[1:5,], page=stdout())
}
```

Which specifies the content type and some HTML text. No headers are needed here. The `d` variable refers to some data frame. If there were an error, we would return an error code, say `404L` for file not found. In this case the error handler is called.

Using `JSON` is not much different. This example will use the package `rjson` to create encode the data into json code, but `RJSONIO` can be used instead (from `omegahat.org`) or one can create the JSON within R directly. Here is the server side code (not written with any generality):

```
show_json <- function() {
  require(rjson, quietly=TRUE)
  n <- as.integer(GET$n)
  n <- min(max(n,1), 32)                # check
  out <- toJSON(list(mpg=mtcars$mpg[1:n],
                     car=rownames(mtcars)[1:n]))
  setContentType("application/json")
  cat(out)
}
```

We allow a variable `n` to be passed in through the Ajax call. The function `toJSON` prefers lists to data frames, so we make a list with our data, in this case we have two named variables `mpg` and `car`.

Within the HTML file we have this JavaScript code.

```
$.getJSON(
  "http://localhost/ajaxapp/json",
  {n:"5"},
  function(data) {
    $("#jsonTarget").html("");  // clear out
    for(i=0; i < data.mpg.length; i=i+1) {
```

```
      $("#jsonTarget").append(data.car[i] + " gets " +
        data.mpg[i] + " miles per gallon" + "<br />");
    }
   });
```

The `getJSON` method is a convenience for the `ajax` method. The second argument is how we pass in the parameter `n`. Finally, the last function is called on a success, and simply loops over the vector and pieces together some HTML, appending it to the target. (The last bit is much easier in R, but not too hard in JavaScript.)

Finally, we illustrate doing a similar task only using XML. The server side code might look like

```
show_xml <- function() {
  require(XML, quietly=TRUE)
  n <- as.integer(GET$n)
  n <- min(max(n,1), 32)                # check
  children <- sapply(1:n, function(i)
                     newXMLNode("car",
                                newXMLNode("make",d[i,1]),
                                newXMLNode("mpg", d[i,2])
                                ))
  out <- saveXML(newXMLNode("data", .children=children))
  setContentType("text/xml")
  cat(out)
}
```

We use the `XML` library to piece together our response. In this case we make several car nodes, each with a make and mpg value.

The JavaScript to parse this response can look like this:

```
$.ajax({
  type: "GET",
  url:"http://localhost/ajaxapp/xml",
  data: {n:"4"},
  dataType: "xml",
  success: function(data) {
    $("#xmlTarget").html("");
    $(data).find("car").each(function() {
     $("#xmlTarget").append($(this).find("make").text()  +
       " gets " + $(this).find("mpg").text() +
       " miles per gallon" + "<br />")
    })
}
})
```

The `data` argument is used to pass in a parameter. As for the `success` callback, as before we append tet to the target after clearing it out. To find the text, is a bit tricky, as it uses `jQuery`'s selector methods. Basically, the vari-

able `this` stands for each `car` node, and the `find` method gets the child node for that variable. The `text` method converts the object to text that can be appended to the target.