6. Web services, XML, JSON.

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2017

HTTP

The `httr`* package does calls to web servers, to help you write interfaces.

Main functions

- `GET()`, `POST()`
- `content()`, `http_status()`, `headers()`

*'hitter', not 'hater'

Simple requests

```
> ok <- GET("http://faculty.washington.edu/kenrice/sisg-adv")
> http_status(ok)$reason
[1] "OK"
> notok<- GET("http://faculty.washington.edu/kenrice/pony")
> http_status(notok)$reason
[1] "Not Found"
> weird <- GET("http://error418.net")
> http_status(weird)$reason
[1] "I'm a teapot (RFC 2324)"
```

Simple processing

```
> content(ok)
{xml_document}
<html>
[1] <head>\n  <title>SISG: Module 17</title>\n  <meta h ...
[2] <body style="padding: 0px; margin: 0px;"> &#13;\n   ...
```

Simple processing

```
> library(xml2)
> as_list(content(ok))$body
...

[2]]
[[2]][[1]]
[[2]][[1]]$td
[[1]]
[1] "\r\n      \r\n\r\n"

[[2]]
[[1]]
[1] "Summer Institute in Statistical Genetics"

attr(,"id")
[1] "header_textarea1_heading"
attr(,"class")
[1] "title contenttitle"
```

More complicated: JSON format

```
> paperquery<-
 GET("https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi?
      db=pmc&id=4780701&retmode=json")
> cat(content(paperquery,"text"))
{
    "header": {
        "type": "esummary",
        "version": "0.3"
    },
    "result": {
        "uids": [
            "4780701"
        ],
        "4780701": {
            "uid": "4780701",
            "pubdate": "2016 Mar 7",
            "epubdate": "2016 Mar 7",
            "printpubdate": "",
            "source": "PLoS One",
            "authors": [
                {
                    "name": "Dehghan A",
                    "authtype": "Author"
```

More complicated: JSON format

```
> names(paper)
[1] "header" "result"
> names(paper$result)
[1] "uids"    "4780701"
> names(paper$header)
[1] "type"    "version"
> names(paper$result)
[1] "uids"    "4780701"
> names(paper$result$`4780701`)
 [1] "uid"             "pubdate"         "epubdate"
 [4] "printpubdate"    "source"          "authors"
 [7] "title"           "volume"          "issue"
[10] "pages"           "articleids"      "fulljournalname"
[13] "sortdate"        "pmclivedate"
> paper$result$`4780701`$title
[1] "Genome-Wide Association Study for Incident Myocardial Infarction
 and Coronary Heart Disease in Prospective Cohort Studies: The CHARGE Consortium"
```

Scripps WELLderly cohort

Sometimes, the package can't guess the format correctly and you have to say

```
sequencequery<-
 GET("https://genomics.scripps.edu/browser/SCN10A/illumina/json")
snps<-content(a,type="application/json")
> length(snps)
[1] 1034
> snps[[1]]
[1] "3,38715075,rs79960161,T,G,999,PASS,SNP;SCN10A;SCN10A(uc003ciq.
2);Protein_Coding;Downstream;.;.;T-0.9540,G-0.0460;T-311,G-15;
T/T-0.9080,T/G-0.0920,G/G-0.0000;T/T-148,T/G-15,G/G-0,N/N-0"
```

Complex text data

XML is a format for constructing and describing data formats for plain text data.

- HTML (almost)
- SVG graphics format
- MS Word and Open Office files
- MAGE-ML for microarray data
- Many web services
- KML for Google Earth

Components of XML

Tag A markup construct that begins with < and ends with >. Tags come in three flavors: start-tags, for example `<section>`, end-tags, for example `</section>`, and empty-element tags, for example `<line-break/>`.

Element A start-tag and matching end-tag and what is between them. The characters between the start- and end-tags,may contain markup, including other elements, which are called child elements. An example of an element is `<Greeting>Hello, world.</Greeting>` Empty-element tags also count as elements: `<line-break/>`.

Attribute A name and value stored within the start tag, eg in HTML `<a href="link.html">` the `href` is an attribute with value `link.html`.

Schemas

It is straightforward to parse any XML document into a tree of elements with no additional information. An XML schema or DTD defines a specific XML language by specifying which tags and attributes are valid, and what nesting is allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <Placemark>
     <name> SCS </name>
     <Description> coffee </Description>
     <Point><coordinates>-122.31067,47.64944,0</coordinates></Point>
    </Placemark>
  </Document>
</kml>
```

Schemas

The KML schema is defined at http://schemas.opengis.net/kml/2.2.0/ogckml22.xsd.

- `kml` tag defines a KML document, `xmlns` defines a namespace, to avoid confusion between different definitions of tags such as `point`

- `placemark` tag defines a geographic location. There are many ways to do this, we have the simplest one
  - `name` tag for name of the place
  - `description` tag for longer description
  - `point` tag for location
    * `coordinates` tag relative to WGS84 reference spheroid.

Making XML

Simple XML such as a single KML point *can* be created just by manipulating strings;

```
kml<-function(conn,lat,lon,name,desc){
     ss<-gsub(" ","",paste("<coordinates>",lon,",",lat,
         ",0</coordinates>"))
     cat("<?xml version=\"1.0\" encoding=\"UTF-8\"?>
        <kml xmlns=\"http://www.opengis.net/kml/2.2\">
   <Document>
       <Placemark>
        <name>",name,"</name>
        <Description>",desc,"</Description>
        <Point>",
        ss,
        " </Point>
       </Placemark>
    </Document>
        </kml>\n",
     file=conn)
}

kml("temp.kml", 47.64944,-122.31067,"SCS", "coffee")
shell("notepad temp.kml")
shell.exec("temp.kml")
```

Making XML

For more complex XML, prefer software that understands the tree structure.

`XML` package provides routines for reading and writing XML

To create XML, use `xmlOutputBuffer` to start a document.

The returned value is an XML object with a list of functions to add tags, close tags, add a whole child subtree, and print out the current state of the document.

[This also shows how to implement a familiar style of object-oriented programming in R]

Making XML

```
con <- xmlOutputBuffer(nsURI="http://www.opengis.net/kml/2.2",
      nameSpace="")

con$addTag("kml",close=FALSE)
  con$addTag("Placemark",close=FALSE)
    con$addTag("name",  "SCS")
    con$addTag("description", "Registration, Coffee, Snacks")
    con$addTag("Point", close=FALSE)
      con$addTag("coordinates", "47.64944,-122.31067,0")
    con$closeTag() # Point
  con$closeTag() # Placemark
con$closeTag() #kml

cat(con$value())
```

SVG

W3C Vector graphics format, allows animations, zooming, etc. Only partial support in most browsers.

R has `svg()` graphics device; `RSVGDevice`, `RSVGToolTips` packages.

```
for(i in 1:length(or)) {
   setSVGShapeToolTip(title=gene[i],
     desc1=snp[i],
     desc2=if(abs(lor[i]/se[i])>qnorm(0.5/n,lower.tail=FALSE))
         qvals[i]
     else
         NULL)
     setSVGShapeURL(paste("http://pga.gs.washington.edu/data",
         tolower(gene[i]),sep="/"))
     points(prec[i],lor[i], cex=1, pch=19, col='grey')
     invisible(NULL)
 }
```

Creates an SVG file showing SNP associations, with clickable links for annotation.

SVG file

File begins

```
<svg version="1.1" baseProfile="full" width="722.70" height="578.16"
viewBox="0,0,722.70,578.16" onload="Init(evt)" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:ev="http://www.w3.org/2001/xml-events">
 <title>R SVG Plot</title>
 <desc>R SVG Plot with tooltips! (mode=2)</desc>
```

specifying the type of XML format and where its specification can be found.

Each point looks like

```
 <a href="http://pga.gs.washington.edu/data/f10">
  <circle cx="333.66" cy="365.76" r="2.34" stroke-width="1px" stroke="#BEBEBE"
              fill="#BEBEBE" stroke-opacity="1.000000" fill-opacity="1.000000">
   <title>F10</title>
   <desc1>rs3211744</desc1>
  </circle>
 </a>
```

Reading XML

XML package has two approaches to reading XML

- whole file into memory at once ('DOM')
- analyse and discard each node as it is read. ('event')

```
library(XML)
funnelplot <- xmlTreeParse("svgplot1.svg", useInternal=TRUE)
```

The `funnelplot` variable now holds the whole XML (SVG) document in a tree structure.

Manipulating XML

Read the x coordinate of each point (the `cx` attribute of the `<circle>` elements, nested in the `<a>` elements, nested in the `<svg>` element)

```
xpathApply(funnelplot, "/s:svg/s:a/s:circle",xmlGetAttr, "cx",
    namespaces=c(s="http://www.w3.org/2000/svg"))
```

`xpathApply` finds nodes in the tree satisfying the path `<svg> <a href> <circle>` and then applies `xmlgetAttr()` to each one, with the argument `cx`, to extract the x-axis coordinate. This returns a list of 171 numbers.

The `s:` prefix refers to the namespace, where the format is defined. The `namespaces` argument lets us abbreviate the full namespace `http://www.w3.org/2000/svg` by `s`.

Manipulating XML

Find the points corresponding to the gene TFPI:

```
xpathApply(funnelplot, "/s:svg/s:a/s:circle[s:title='TFPI']",
    namespaces=c(s="http://www.w3.org/2000/svg"))
```

`xpathApply` finds nodes in the tree satisfying the path `<svg> <a href> <circle>` and the condition `title='TFPI'`. Since we do not specify a function to apply to each node, a list of nodes is returned.

Example: CRANberries

CRANberries is an RSS feed that describes new and updated packages on CRAN.

These `XML` commands can be used to read the file, and count the number of new packages in the past week or so;

```
cr  <- readLines("http://dirk.eddelbuettel.com/cranberries/index.rss",
     encoding="ISO-8859-1")
cr <- iconv(cr, to="UTF-8")
crt <- xmlTreeParse(cr, useInternal=TRUE)
titles <-xpathApply(crt, "/rss/channel/item/title")
titlelist <- sapply(titles, xmlToList)
titlelist

length(grep("New package", titlelist)) # how many new ones?
```

biomaRt

Bioconductor's biomaRt package sends queries to the Ensembl database server (www.ensembl.org). Ensembl stores genome annotation data for about 50 species, including cross-species information.

A biomaRt operation like:

```
ens <- useMart("ensembl")
#listDatasets(ens)
human <- useDataset("hsapiens_gene_ensembl",mart=ens)
mouse <- useDataset("mmusculus_gene_ensembl",mart=ens)
getLDS(attributes = c("hgnc_symbol","chromosome_name",
           "start_position"),
         filters = "hgnc_symbol", values = "NOX1",
         mart = human,
       attributesL =c("chromosome_name","start_position"),
         martL = mouse)
```

translates to

biomaRt

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Query>
<Query  virtualSchemaName = 'default' uniqueRows = '1'
         count = '0' datasetConfigVersion = '0.6'
         requestid= "biomaRt">
   <Dataset name = 'hsapiens_gene_ensembl'>
     <Attribute name = 'hgnc_symbol'/>
     <Attribute name = 'chromosome_name'/>
     <Attribute name = 'start_position'/>
     <Filter name = 'hgnc_symbol' value = 'NOX1' />
   </Dataset>
   <Dataset name = 'mmusculus_gene_ensembl' >
      <Attribute name = 'chromosome_name'/>
      <Attribute name = 'start_position'/>
   </Dataset>
</Query>
```

biomaRt

The `biomaRt` package constructs the XML by pasting character strings rather than with the `XML` package.

It uses the `RCurl` package to post the XML request to the webserver and collect the response, which is a table of strings.