Quick Guide to XML in R

Jesse Lecy

July 7, 2016

The IRS has released data on 990 tax returns from electronic nonprofit and foundation filers. The data is reported as XML documents for each return.

In order to make the data accessible, we are working to

BASIC DATA PRINCIPLES

Data used for statistical analysis often comes in flat, two-dimensional spreadsheets. Rows represent observations, and columns represent variables.

Out in the real world, however, data is often generated through relational databases that can capture the one-to-many relationships inherent in systems.

As an example, if you have shopped on Amazon your customer information has been stored in their database, and information about each transaction has been captured. They use two different tables for this information - one for customers and one for transactions. If they tried to store all of this information in a single spreadsheet much of the static information (your name and address) would have to be repeated. Since they have billions of transactions, each with hundreds of fields, that spreadsheet would quickly become large and slow (computationally expensive, as they say in the industry).

Relational databases - storing information that has a one-to-many relationship in different tables - simplifies life and makes for robust systems. If the information needs to be combined, for example for market analysis to look at the relationship between products and zip codes, it can be done through a query.

```
customer.info
```

```
##   CUSTOMER.ID FIRST.NAME LAST.NAME       ADDRESS ZIP.CODE
## 1         178     Alvaro    Jaurez  123 Park Ave    57701
## 2         934    Janette   Johnson  456 Candy Ln    57701
## 3         269    Latisha     Shane 1600 Penn Ave    20500
```

```
purchases
```

```
##   CUSTOMER.ID PRODUCT PRICE
## 1         178   video  5.38
## 2         178  shovel 12.00
## 3         269    book  3.99
## 4         269   purse  8.00
## 5         934  mirror  7.64
```

```
merge( customer.info, purchases )
```

```
##   CUSTOMER.ID FIRST.NAME LAST.NAME       ADDRESS ZIP.CODE PRODUCT PRICE
## 1         178     Alvaro    Jaurez  123 Park Ave    57701   video  5.38
## 2         178     Alvaro    Jaurez  123 Park Ave    57701  shovel 12.00
## 3         269    Latisha     Shane 1600 Penn Ave    20500    book  3.99
## 4         269    Latisha     Shane 1600 Penn Ave    20500   purse  8.00
## 5         934    Janette   Johnson  456 Candy Ln    57701  mirror  7.64
```

XML DATA MODELS

The world of web programming requires a different way of representing data. If we want to build smart websites, browsers and web applications need to be able to make sense of data. As a result, eXtensible Markup Langauge was invented in order to (1) provide context for informaton so it can be made more useful, and (2) provide a flat, linear representation of relational databases.

XML works by adding tags to information so that computers can make sense of it:

```
<ANIMAL>DOG</ANIMAL>

<PLANT>FLOWER</PLANT>
```

This scheme also allows for the creation of sets of things.

```
<ANIMALS>

   <FISH>Tuna</FISH>
   <MAMMAL>Dog</MAMMAL>
   <SNAKE>Viper</SNAKE>

</ANIMALS>
```

We can now describe the world in a couple of different ways - we can ask for the set of all animals, or we can ask for specific types of animals.

Similarly, here is what part of the relational database above would look like.

```
<MEMBERS>

   <CUSTOMER>
      <ID>178</ID>
      <FIRST.NAME>Alvaro</FIRST.NAME>
      <LAST.NAME>Juarez</LAST.NAME>
      <ADDRESS>123 Park Ave</ADDRESS>
      <ZIP>57701</ZIP>
   </CUSTOMER>

   <CUSTOMER>
      <ID>934</ID>
      <FIRST.NAME>Janette</FIRST.NAME>
      <LAST.NAME>Johnson</LAST.NAME>
      <ADDRESS>456 Candy Ln</ADDRESS>
      <ZIP>57701</ZIP>
   </CUSTOMER>

</MEMBERS>

<PURCHASES>

   <TRANSCTION>
      <ID>178</ID>
      <PRODUCT>video</PRODUCT>
      <PRICE>5.38</PRICE>
   </TRANSACTION>

   <TRANSCTION>
      <ID>178</ID>
      <PRODUCT>shovel</PRODUCT>
      <PRICE>12.00</PRICE>
   </TRANSACTION>

</PURCHAES>
```

ACCESSING DATA WITH XML

Accessing data in the XML format requires a basic understanding of two principles - nodes and paths.

Let's return to the example above:

```
<MEMBERS>

   <CUSTOMER>
      <ID>178</ID>
      <FIRST.NAME>Alvaro</FIRST.NAME>
      <LAST.NAME>Juarez</LAST.NAME>
      <ADDRESS>123 Park Ave</ADDRESS>
      <ZIP>57701</ZIP>
   </CUSTOMER>

   <CUSTOMER>
      <ID>934</ID>
      <FIRST.NAME>Janette</FIRST.NAME>
      <LAST.NAME>Johnson</LAST.NAME>
      <ADDRESS>456 Candy Ln</ADDRESS>
      <ZIP>57701</ZIP>
   </CUSTOMER>

</MEMBERS>
```

Here each tag represents a separate node, so this XML document contains three layers of nodes - members, customers, and the set of nodes related to customer information.

You can start to see that the nested nature of data creates a tree structure. Since it is hierarchical, each **parent** node contains **children** nodes. MEMBERS contains CUSTOMER nodes. CUSTOMER contains ID, NAME, and ADDRESS nodes.

To manipulate the data, we can grab a single node at once, and work with all of its sub-elements.

We will use the *xml2* package and the *xml_parent()* and *xml_children()* functions. The function *xml_name()* prints the name of the node.

```r
# install.packages( "xlm2" )

library( xml2 )
```

```
## Warning: package 'xml2' was built under R version 3.3.1
```

```
dat <- read_xml( "<MEMBERS>
                     <CUSTOMER>
                        <ID>178</ID>
                        <FIRST.NAME>Alvaro</FIRST.NAME>
                        <LAST.NAME>Juarez</LAST.NAME>
                        <ADDRESS>123 Park Ave</ADDRESS>
                        <ZIP>57701</ZIP>
                     </CUSTOMER>
                     <CUSTOMER>
                        <ID>934</ID>
                        <FIRST.NAME>Janette</FIRST.NAME>
                        <LAST.NAME>Johnson</LAST.NAME>
                        <ADDRESS>456 Candy Ln</ADDRESS>
                        <ZIP>57701</ZIP>
                     </CUSTOMER>
                  </MEMBERS>"  )


xml_name(  dat  )
```

```
## [1] "MEMBERS"
```

```
xml_name( xml_parent( dat ) )  # no parents - it is a root node
```

```
## [1] ""
```

```
xml_name( xml_children( dat ) )
```

```
## [1] "CUSTOMER" "CUSTOMER"
```

We can break the document apart into individual nodes, and treat them separately using *xml_find_first()* and *xml_find_all()*.

```
customer1 <- xml_find_first( dat, "//CUSTOMER" )

xml_name(  customer1  )
```

```
## [1] "CUSTOMER"
```

```
xml_name( xml_parent( customer1 ) )
```

```
## [1] "MEMBERS"
```

```
xml_name( xml_children( customer1 ) )
```

```
## [1] "ID"         "FIRST.NAME" "LAST.NAME"  "ADDRESS"    "ZIP"
```

And finally, we can access data within a “leaf” node using the *xml_text()*, *xml_double()*, and *xml_integer()* commands.

```
xml_text( xml_find_first( dat, "//ADDRESS" ) )
```

```
## [1] "123 Park Ave"
```

```
xml_text( xml_find_all( dat, "//ADDRESS" ) )
```

```
## [1] "123 Park Ave" "456 Candy Ln"
```

Putting it together, we can start to build datasets:

```
first.name <- xml_text( xml_find_all( dat, "//FIRST.NAME" ) )
last.name <- xml_text( xml_find_all( dat, "//LAST.NAME" ) )
address <- xml_text( xml_find_all( dat, "//ADDRESS" ) )
zip <- xml_text( xml_find_all( dat, "//ZIP" ) )

data.frame( first.name, last.name, address, zip )
```

```
##   first.name last.name      address   zip
## 1     Alvaro    Juarez 123 Park Ave 57701
## 2    Janette   Johnson 456 Candy Ln 57701
```

NAME SPACES

The header information in an XML document - the root node - may contain information on "name spaces".

```
# contains no name spaces

<Return returnVersion="2014v5.0">


# contains name spaces

<Return xmlns="http://www.irs.gov/efile" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schen
```

This topic is beyond this short intro to XML, other than to note you do not need them to access the data and they will give you a headache, so remove them using the *xml_ns_strip()* function.

```
# LOAD DATA FROM A PC FILING FULL 990

dat <- read_xml( x="https://s3.amazonaws.com/irs-form-990/201541349349307794_public.xml", options=NULL

# STRIP THE NAMESPACE

xml_ns_strip( dat )
```

XPATH

In the XML structure there is a specific convention for referencing nodes called *xpath*. You can create sophisticated statements to access specific elements of your data, but the basic principles of a search statement are:

- Place two backslashes in front of the top node in your search
- Separate subsequent nodes with a single backslash
- Add a period if you want the search to be local

Node that arguments to the search functions *xml_find_all()* and *xml_find_first()* are a nodeset, and an xpath.

```
xml_text( xml_find_first( dat, "//ADDRESS" ))
```

```
## [1] "123 Park Ave"
```

```
# if the ADDRESS field name is used in other parts of the data
# you will need to be more specific

xml_text( xml_find_first( dat, "//CUSTOMER/ADDRESS" ))
```

```
## [1] "123 Park Ave"
```

```
xml_text( xml_find_first( dat, "//MEMBERS/CUSTOMER/ADDRESS" ))
```

```
## [1] "123 Park Ave"
```

```
# drilling down into a node

xml_text( xml_find_all( dat, "//ADDRESS" )) # returns all customer addresses
```

```
## [1] "123 Park Ave" "456 Candy Ln"
```

```
customer1 <- xml_find_first( dat, "//CUSTOMER" )  # create node for first customer only

xml_text( xml_find_all( customer1, ".//ADDRESS" ))
```

```
## [1] "123 Park Ave"
```

```
# note the use of the period in the xpath!

xml_text( xml_find_all( customer1, ".//ADDRESS" )) # with period search is local to nodeset
```

```
## [1] "123 Park Ave"
```

```
xml_text( xml_find_all( customer1, "//ADDRESS" )) # no period still searches full document
```

```
## [1] "123 Park Ave" "456 Candy Ln"
```

You can identify specific paths using the *xml_path()* function. This will give you a way to reference each non-unique node in the path individually.

```r
xml_path( xml_find_all( dat, "//ADDRESS") )
```

```
## [1] "/MEMBERS/CUSTOMER[1]/ADDRESS" "/MEMBERS/CUSTOMER[2]/ADDRESS"
```

```r
xml_text( xml_find_all( dat, "//MEMBERS/CUSTOMER[1]/ADDRESS" ))
```

```
## [1] "123 Park Ave"
```

```r
xml_text( xml_find_all( dat, "//MEMBERS/CUSTOMER[2]/ADDRESS" ))
```

```
## [1] "456 Candy Ln"
```

XML ATTRIBUTES

Some nodes contain attributes that store meta-data associated with the node:

```
# contains no attributes

<ReturnData> ... </ReturnData>

<IRS990> ... </IRS990>

# contains attributes

<ReturnData documentCnt="6"> ... </ReturnData>

<IRS990 referenceDocumentId="RetDoc1044400001"> ... </IRS990>
```

We can access meta-data using the *xml_attr()* function.

```r
# LOAD DATA FROM A PC FILING FULL 990

dat <- read_xml( x="https://s3.amazonaws.com/irs-form-990/201541349349307794_public.xml", options=NULL

# STRIP THE NAMESPACE

xml_ns_strip( dat )


# GRAB THE ATTRIBUTES

return.data <- xml_find_first( dat, "//ReturnData" )  # grab return data node

xml_attrs( x=return.data )  # list attributes for node
```

```
## documentCnt
##         "6"
```

```
xml_attr( x=return.data, attr="documentCnt" )  # return document count attribute
```

```
## [1] "6"
```

990 DATA

Now for applying these rules to the data at hand. If we look at the structure of a 990 return, it looks something like this:

```
990 RETURN

   HEADER DATA

      NAME, EIN, YEAR, etc.

   RETURN DATA

      REVENUES
      EXPENSES
      GOVERNANCE

        BOARD MEMBERS

      FUNCTIONAL REVENUE CATEGORIES
      FUNCTIONAL EXPENSE CATEGORIES

      SCHEDULES

        A-Public Support
        B-Contributors
        D-Supplemental Financial Statements
        M-Non-Cash Contributions
        O-Supplemental Information
        R-Related Organizations

END 990 RETURN
```

Note that the structure varies between the 990, 990-EZ, and 990-PF returns, and that schedules included will vary by organization and year.

If you would like to see an example of the full structure, run this code (not executed here because the output is long):

```
# library( xml2 )
#
# xml_structure( read_xml( x="https://s3.amazonaws.com/irs-form-990/201541349349307794_public.xml", opt
```

Here is the example organization listed on on the IRS Amazon Web Server page:

https://aws.amazon.com/public-data-sets/irs-990/

```r
# LOAD DATA FROM A PC FILING FULL 990

dat <- read_xml( x="https://s3.amazonaws.com/irs-form-990/201541349349307794_public.xml", options=NULL 

# STRIP THE NAMESPACE

xml_ns_strip( dat )



# EXAMINE THE DATA

# xml_structure( dat )  # this is overwhelming, we need to be able to drill down

xml_name( xml_root( dat ) )  # only one root node that contains all data
```

```
## [1] "Return"
```

```r
xml_name( xml_children( dat ) )  # split into two sections, header and return data
```

```
## [1] "ReturnHeader" "ReturnData"
```

```r
xml_name( xml_children( xml_find_first( dat, "//ReturnHeader" ) ) )
```

```
##  [1] "ReturnTs"           "TaxPeriodEndDt"     "PreparerFirmGrp"
##  [4] "ReturnTypeCd"       "TaxPeriodBeginDt"   "Filer"
##  [7] "BusinessOfficerGrp" "PreparerPersonGrp"  "TaxYr"
## [10] "BuildTS"
```

```r
xml_name( xml_children( xml_find_first( dat, "//ReturnData/IRS990" ) ) )[ 1:10 ]
```

```
##  [1] "PrincipalOfficerNm"          "USAddress"
##  [3] "GrossReceiptsAmt"            "GroupReturnForAffiliatesInd"
##  [5] "Organization501c3Ind"        "WebsiteAddressTxt"
##  [7] "TypeOfOrganizationCorpInd"   "FormationYr"
##  [9] "LegalDomicileStateCd"        "ActivityOrMissionDesc"
```

```r
xml_text( xml_children( xml_find_first( dat, "//ReturnData/IRS990" ) ) )[ 1:10 ]
```

```
##  [1] "SCOTT LEWIS"
##  [2] "\n        2508 HISTORIC DECATUR SUITE 120\n        SAN DIEGO\n        CA\n        92106\n
##  [3] "1753504"
##  [4] "0"
##  [5] "X"
##  [6] "VOICEOFSANDIEGO.ORG"
##  [7] "X"
##  [8] "2004"
##  [9] "CA"
## [10] "ON-LINE NEWSPAPER OPERATED EXCLUSIVELY TO EDUCATE AND INFORM RESIDENTS OF SAN DIEGO COUNTY THR
```

You can see that any node that contains children (for example USAddress) will print the data separated by a carriage return (\n).

As an example of how you might parse the data to create a dataset:

```
header.data <- xml_find_first( dat, "//ReturnHeader" )


TaxYr <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/TaxYr" ) )
ReturnType <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/ReturnTypeCd" ) )
EIN <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/EIN" ) )
BusinessName <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/BusinessName/BusinessNameLine
Address <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/USAddress/AddressLine1Txt" ) )
City <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/USAddress/CityNm" ) )
State <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/USAddress/StateAbbreviationCd" ) )
Zip <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/Filer/USAddress/ZIPCd" ) )
TaxPrep <- xml_text( xml_find_all( dat, "//Return/ReturnHeader/BusinessOfficerGrp/DiscussWithPaidPrepar


header.df <- data.frame( TaxYr, ReturnType, EIN, BusinessName, Address, City, State, Zip, TaxPrep )

header.df
```

```
##   TaxYr ReturnType       EIN       BusinessName
## 1  2014        990 201585919 VOICE OF SAN DIEGO
##                           Address      City State   Zip TaxPrep
## 1 2508 HISTORIC DECATUR SUITE 120 SAN DIEGO    CA 92106       1
```

To build a full dataset, you will need to iterate over multiple 990 returns and combine these data. This would be approximately:

```
library( dplyr )

dat1 <- read_xml( url.01 )

# create header.df.01

dat2 <- read_xml( url.02 )

# create header.df.02

bind_rows( header.df.01, header.df.02 )
```