Web Technologies in R

A Short Introduction to Web Technologies in R

Gaston Sanchez

November 2022

Contents

About

In this manuscript I provide an overview of some of the Web Technologies available in R.

About You

I am assuming two things about you. In decreasing order of importance:

1) You already know R—this is not an introductory text on R—.

2) You already use R for handling quantitative and qualitative data, but not (necessarily) for working with data from Web.

Citation

You can cite this work as:

Sanchez, G. (2022) A Short Introduction to Web Data Technologies in R. https://www.gastonsanchez.com/R-web-technologies

---

My Series of R Tutorials

This document is part of a series of texts that I've written about Programming and Data Analysis in R:

- **Breaking the Ice with R: Getting Started with R and RStudio**

https://www.gastonsanchez.com/R-ice-breaker

- **Tidy Hurricanes: Analyzing Tropical Storms with Tidyverse Tools**

https://www.gastonsanchez.com/R-tidy-hurricanes

- **R Coding Basics: An Introduction to the Basics of Coding in R**

https://www.gastonsanchez.com/R-coding-basics

- **Rolling Dice: Exploring Simulations in Games of Chance with R**

https://www.gastonsanchez.com/R-rolling-dice

- **Web Tech in R: A Short Introduction to Web Technologies in R**

https://www.gastonsanchez.com/R-web-technologies

Donation

As a Data Science and Statistics educator, I love to share the work I do. Each month I spend dozens of hours curating learning materials like this resource. If you find any value and usefulness in it, please consider making a one-time donation—via paypal—in any amount (e.g. the amount you would spend inviting me a cup of coffee or any other drink). Your support really matters.

License

Part I

Intro

Chapter 1

Introduction

The Web is full of information and resources that can be considered to be sources of data. Statisticians, data analysts, data scientists, researchers, and data-based users in general, increasingly are working in projects that depend on various data sources, many of them either coming or available from the Web.

As it turns out, we can use a wide array of approaches to get data from the Web. For instance, we can simply scrape data from human-readable webpages. Likewise, we can also utilize application programming interfaces (APIs) to request some data sets. Interestingly, the data may come in some XML dialect, the most common one being HTML. But it can also come in a JSON document or some other self-describing format. Consequently, you need to be prepared to deal with data from the Web.

1.1 Suggested Tools

To enjoy the content of this text, and also to be able to replicate the examples discussed in subsequent chapters, you will need the following tools:

- A fairly recent version of R
- A fairly recent version of RStudio
- Web Browser (e.g. Chrome, Safari, Firefox, Opera)
- and good Internet connection!

1.2 Suggested R Packages

The code and examples shown in this book are based on the following packages:

- `"tidyverse"` which contains, among other packages:

  - `"dplyr"`: for manipulation of data tables
  - `"stringr"`: for manipulation of strings and text data
- `"xml2"`: tools for parsing XML and HTML documents
- `"httr"`: tools for working with HTTP requests
- `"rvest"`: for *harvesting* or scraping web data in an easy way
- `"jsonlite"`: functions for handling JSON data

By the way, R has a large collection of packages for interacting with the Web. A comprehensive list of packages for dealing with *Web Technologies* is available in the following Cran Task View (curated by Mauricio Vargas Sepulveda):

https://cran.r-project.org/web/views/WebTechnologies.html

1.3 Some Acronyms

As you'll see later in this book, there is a number of acronyms commonly used around all-things Web. I will define and explain every acronym in their corresponding chapter. In the meantime, I would like to give you a first exposure to the following terms:

- **WWW**: World Wide Web
- **URL**: Uniform Resource Locator
- **HTTP**: HyperText Transfer Protocol
- **XML**: Extensible Markup Language
- **HTML**: HyperText Markup Language
- **JSON**: JavaScript Object Notation
- **API**: Application Programming Interface

Chapter 2

The Web

In this chapter I provide a brief and superficial description of the Web and how it works.

2.1 Surfing the Web

Think about when you surf the web:

- You open a web browser (e.g. Google Chrome, Safari, Firefox)
- You type in or click the URL of a website you wish to visit (e.g. https://www.r-project.org)
- You wait some fractions of a second, and then the website shows up in your screen.

What exactly is happening "behind the scenes"?

- People access websites using software called a **Web browser** (e.g. Google Chrome, Safari, Firefox)
- A browser is a software that, among other things, **requests** information (e.g. request to access R project's website)
- Using more proper language, the browser in your computer is the **client that requests** a variety of resources (e.g. pages, images, videos, audio, scripts)
- The client's request is sent to **Web servers**
- A **server** is the software-computer in charge of **serving** the resources that the clients request.
- The server **sends responses** back to the client

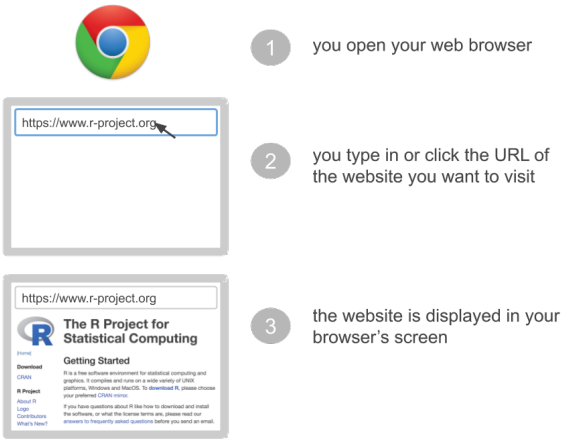


Figure 2.1: Surfing the web

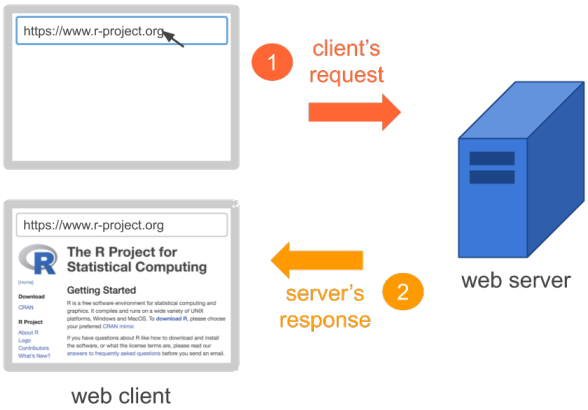


Figure 2.2: Client makes a request, and the serve responds

To be more accurate, the server is the software that allows the computer to communicate with other computers; however, it is common to use the term "server" to refer to the computer running the software, which also contains other files and programs. Simply put, a server is basically a computer connected to the Internet. The Internet, in turn, is just a network of connected computers forming a system of standards and rules. The purpose of connecting computers together is to share information.

The job of the server software is to wait for a request for information, then retrieve and send that information back to the client(s) as fast as possible. In other words, Web servers have a full time job, waiting for requests from Web browsers all over the world.

2.2 How Does the Web Work?

Now that we have the high level intuition of clients making requests, and servers sending responses back to clients, let's describe things in more detail.

To make web pages, programmers, developers and designers create files written in a special type of syntax called **HyperText Markup Language** or HTML for short. These files are stored in a Web server.

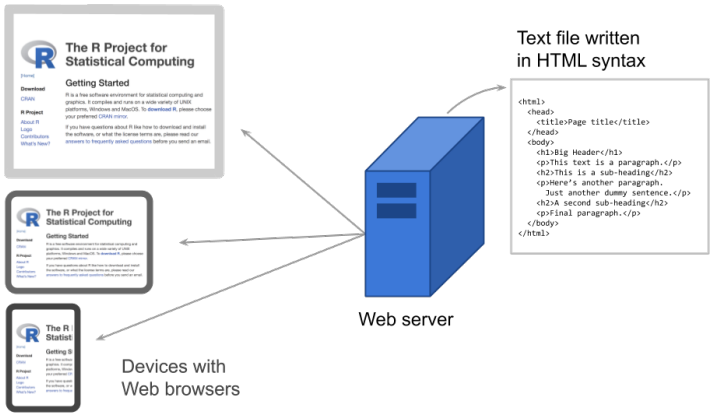


Figure 2.3: HTML files are the building blocks of web pages

To be more precise, Web servers store more than one single HTML file. In practice, websites are made of several directories containing various types of files (image files, audio files, video files, scripts, etc).

Once HTML files are put on the web server, any browser (e.g. Chrome, Safari, Firefox, Explorer) can retrieve the web page over the internet. The browser on your laptop, on your tablet, on your cellphone, you name it. As long as the device you are using is connected to the internet, the browser will retrieve the

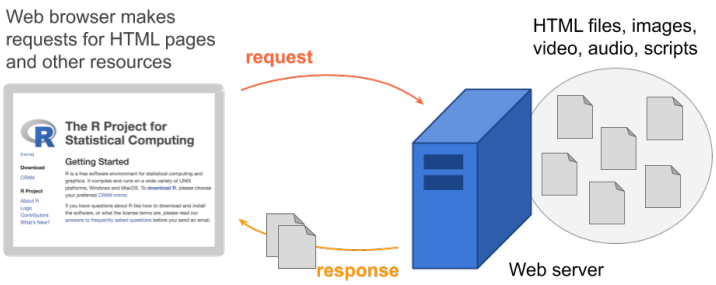


Figure 2.4: Web server containing several types of files, not just HTML files

web page. The HTML content in the web page tells the browser everything it needs to know to display the page.

Figure 2.5: Diagram of the Web

On a side note, it's important to distinguish the Internet from the Web. The Web, originally called the World Wide Web, is just one option to share information over the Internet. What characterizes the Web is that it allows documents to be linked to one another using *hypertext* links or hyperlinks, thus forming a web of interconnected resources.

In Summary

- The Web is a massive distributed information system connecting software and computers to share information.
- The software and computers that form the Web are divided into two types: *clients* and *servers*.
- The way clients and servers dialogue between each other is by following formal **protocols of communication**.

- The main type of protocol that clients and servers use is the HyperText Transfer Protocol (HTTP).
- But there are other ways in which computers can exchange information such as email, file transfer (FTP), and many others.

Chapter 3

Basics of HTTP

In the preceding chapter you were given a high-level description about how the Web works. In this chapter we take the next step to give you a basic introduction to HTTP which is the protocol that servers and browsers use to communicate and exchange information on the Web.

3.1 What is HTTP?

HTTP is the acronym for *Hypertext Transfer Protocol*. Perhaps the two most important terms in this name are *Protocol* and *Transfer*.

According to the dictionary, a **protocol** is:

> "A system of rules that explain the correct conduct and procedures to be followed in formal situations"

In turn, a **transfer protocol** is a communications protocol:

> "A communications protocol is a system of digital rules for data exchange within or between computers."

So, what is HTTP? Simply put, HTTP is the set of rules for transferring things such as text, images, sound, video and other multimedia files over the Web.

3.2 A quick introduction to HTTP

- Whenever you surf the web, your browser sends **HTTP request messages**
- the HTTP requests are sent to Web servers
- web servers handle these requests by returning **HTTP response messages**

- the messages contain the requested resource(s)

3.2.1 HTTP Example

Suppose we open the browser in order to visit R project's homepage

`https://www.r-project.org`

Although we don't see it, there's is a client-server dialogue taking place, illustrated in the diagram below:

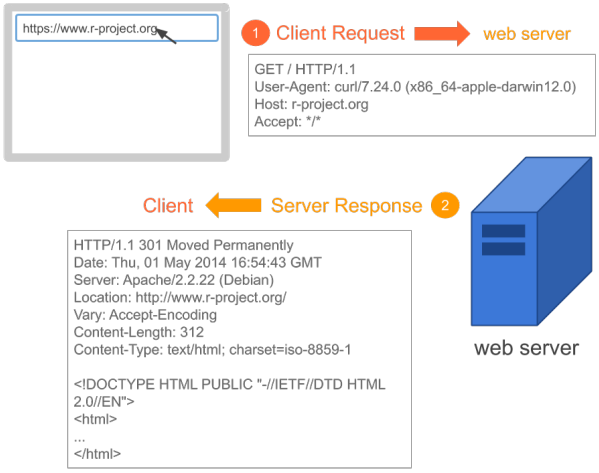


Figure 3.1: Client makes a request, and the serve responds

Using Chrome's **DevTools** (developer tools), we can see the associated information related to the HTTP "conversation" between the client and the server. We provide the content of this dialogue in the following block:

Think of an HTTP request as a set of information sent to the server. When the server receives the request, it (the server) processes the information and provides a response back to the client.

When you visit a URL in your web browser, say R's project website (`https://www.r-project.org`), an HTTP request is made and the response is rendered by the browser as the website you see. Although we don't see the "dialogue" between client and server, it is possible to inspect this interaction using the development tools in a browser such as Chrome's DevTools (like the screenshot above).

The above is a screen-capture in which we can see that the request is composed of a URL (R's project website), and a request method (`GET`) which is what the browser employs to access a website.

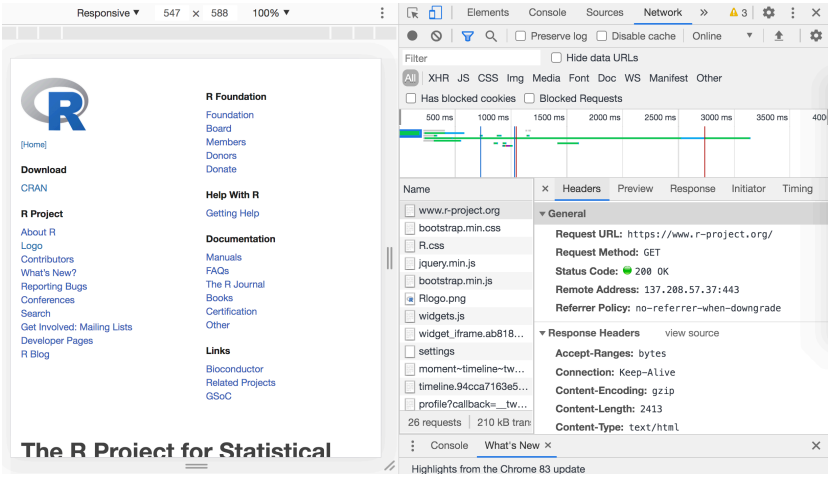


Figure 3.2: Inspecting HTTP messages via DevTools

Figure 3.3: DevTools menu tab for inspecting HTTP messages

3.2.2 HTTP Request

There are several components of an HTTP request (see figure below), but we will focus on the most relevant:

- `URL`: the address or endpoint for the request
- HTTP method or verb: a specific method invoked on the endpoint (`GET`, `POST`, `DELETE`, `PUT`)
- Headers: additional data sent to the server, such as who is making the request and what type of response is expected
- Body: data sent to the server outside of the headers, common for `POST` and `PUT` requests

Figure 3.4: HTTP request headers

3.2.3 HTTP Response

The response headers include the HTTP status code that informs the client how the request was received. There are also other details about the content delivered by the server. In the above example accessing www.r-project.com, we can see the status code success `200`, along with other details about the response content. Notice that the returned content is HTML. This HTML content is what the browser renders into a webpage.

3.3 Anatomy of an HTTP message

HTTP messages consist of 2 parts (separated by a blank line)

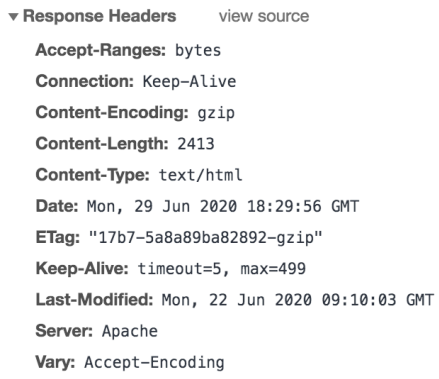


Figure 3.5: HTTP response headers

1) A message **header**
   - the first line in the header is the request/response line
   - the rest of the lines are *headers* formed of `name:value` pairs
2) An optional message **body**

The client (your browser) sends a request to the server:

```
GET / HTTP/1.1
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8y zlib/1.2.5
Host: r-project.org
Accept: */*
```

- The first line is the **request line** which contains: `GET / HTTP/1.1`
- The rest of the *headers* are just `name:value` pairs, e.g. `Host: r-project.org`

The server sends a **response** to the client:

```
HTTP/1.1 301 Moved Permanently
Date: Thu, 01 May 2014 16:54:43 GMT
Server: Apache/2.2.22 (Debian)
Location: http://www.r-project.org/
Vary: Accept-Encoding
Content-Length: 312
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
...
</html>
```

- The first line is the **status line** which contains: `GET / HTTP/1.1`
- The next lines contain **header values**

- The **body** message appears after the blank line, in this case is the content of the HTML page

3.3.1 HTTP Methods

Here's a table with HTTP methods, and their descriptions

Method	Description
`GET`	retrieves whatever information is identified by the Request-URI
`POST`	request with data enclosed in the request body
`HEAD`	identical to GET except that the server MUST NOT return a message-body in the response
`PUT`	requests that the enclosed entity be stored under the supplied Request-URI
`DELETE`	requests that the origin server delete the resource identified by the Request-URI
`TRACE`	invokes a remote, application-layer loop-back of the request message
`CONNECT`	for use with a proxy that can dynamically switch to being a tunnel

So far we've seen that:

- The HTTP protocol is a standardized method for transferring data or documents over the Web
- The clients' requests and the servers' responses are handled via the HTTP protocol
- There are 2 types of HTTP messages: **requests** and **responses**
- We don't actually see HTTP messages but they are there behind the scenes

Part II

XML

Chapter 4

Basics of XML

The goal of this chapter is to give you a crash introduction to XML so that you can get a good grasp of this format for the rest of the book.

- Large amounts of data and information are stored, shared and distributed using XML-dialects.
- They are widely adopted and used in many applications.
- Working with data from the Web often means dealing with some kind of XML dialect.

4.1 What is XML?

XML stands for *eXtensible Markup Language*

Let's dissect the meaning of this acronym. On one hand, XML is a markup language. which means, XML defines a set of rules for encoding information in a format that is both human-readable and machine-readable.

Compared to other types of markup languages (e.g LaTeX, Markdown), XML is used to describe data. To be more precise, XML is a standard for the semantic, **hierarchical** representation of data. This is an important aspect of XML and any of its dialects, because data is represented following a hierarchy.

For instance, one way to organize data is in a table. Conceptually, all elements are stored in cells of a grid structure of rows and columns. Another way to organize data is with hierarchies, that can be visually represented with tree like structures. This latter form of organizing data is what XML uses.

The second aspect, "extensible", means that we can define any number of new formats to represent any kind of data. Therefore, it is extensible. This is a very

interesting aspect of XML because it provides a flexible framework to create new formats for describing and representing data.

Comments

Before moving on, we want to clarify some key terms.

A **markup** is a sequence of characters or other symbols inserted at certain places in a document to indicate either:

- how the content should be displayed when printed or in screen
- describe the document's structure

A Markup Language is a system for annotating (i.e. marking) a document in a way that the content is distinguished from its representation (e.g. LaTeX, PostScript, HTML, SVG)

4.1.1 Marks in XML

In XML (as well as in HTML) the marks (also known as tags) are defined using angle brackets: `< >`.

For example:

```
<mark>Text marked with special tag</mark>
```

The concept of extensibility means that we can define our own marks, the order in which they occur, and how they should be processed. For example we could define marks such as:

- `<my_mark>`
- `<awesome>`
- `<boring>`
- `<pathetic>`

Before moving on, we should mention that XML is NOT:

- a programming language
- a network transfer protocol
- a database

Instead, XML is:

- more than a markup language
- a generic language that provides structure and syntax for representing any type of information
- a meta-language: it allows us to create or define other languages

Here are some famous examples of XML dialects:

- **KML** (Keyhole Markup Language) for describing geo-spatial information used in Google Earth, Google Maps, Google Sky

- **SVG** (Scalable Vector Graphics) for visual graphical displays of two-dimensional graphics with support for interactivity and animation
- **PMML** (Predictive Model Markup Language) for describing and exchanging models produced by data mining and machine learning algorithms
- **RSS** (Rich Site Summary) feeds for publishing blog entries
- **SDMX** (Statistical Data and Metadata Exchange) for organizing and exchanging statistical information
- **SBML** (Systems Biology Markup Language) for describing biological systems

4.1.2 Minimalist Example

Let's consider a handful of XML examples using one of my favorite movies: *Good Will Hunting*, a 1997 American psychological drama film directed by Gus Van Sant, and written by Ben Affleck and Matt Damon.

Figure 4.1: Good Will Hunting (Directed by Gus Van Sant, 1997)

Ultra Simple example

Let's see an ultra simple XML example:

```
<movie>
  Good Will Hunting
</movie>
```

- one single element *movie*
- start-tag: `<movie>`
- end-tag: `</movie>`

- content: `Good Will Hunting`

Elements with attributes

XML elements can have attributes, for example:

```
<movie mins="126" lang="en">
  Good Will Hunting
</movie>
```

- attributes: `mins` (minutes) and `lang` (language)
- attributes are attached to the element's start tag
- attribute values must be quoted!

Elements within other elements

XML elements may contain other elements, for example:

```
<movie mins="126" lang="en">
  <title>Good Will Hunting</title>
  <director>Gus Van Sant</director>
  <year>1998</year>
  <genre>drama</genre>
</movie>
```

- an xml element may contain other elements
- *movie* contains several elements: *title*, *director*, *year*, *genre*

More Embedded elements

As you can tell, the xml element *movie* has a now a hierarchy. We can make it more interesting by including more elements inside *director*.

```
<movie mins="126" lang="en">
  <title>Good Will Hunting</title>
  <director>
    <first_name>Gus</first_name>
    <last_name>Van Sant</last_name>
  </director>
  <year>1998</year>
  <genre>drama</genre>
</movie>
```

Formally, we say that *director* has two child elements: `first_name` and `last_name`.

Tree Structure in XML

We can graphically display the structure of an XML document with a tree diagram, like the following one:

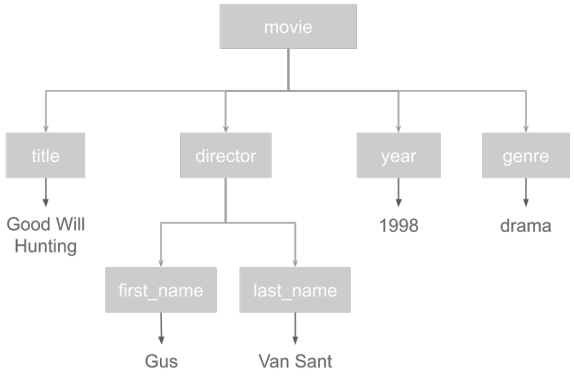


Figure 4.2: XML tree structure

- An XML document can be represented with a **tree structure**
- An XML document must have **one single Root** element
- The `Root` may contain `child` elements
- A `child` element may contain `subchild` elements

4.1.3 Well Formedness

We say that an XML document is **well-formed** when it obeys the basic syntax rules of XML. Some of those rules are:

- one root element containing the rest of elements
- properly nested elements
- self-closing tags
- attributes appear in start-tags of elements
- attribute values must be quoted
- element names and attribute names are case sensitive

Does it matter if an XML document is not Well-formed? Not well-formed XML documents produce potentially fatal errors or warnings when parsed.

Keep in mind that documents may be well-formed but not valid. Well-formed just guarantees that the document meets the basic XML structure, not that the content is valid.

4.1.4 Additional XML Elements

Some Additional Elements

```
<?xml version="1.0"? encoding="UTF-8" ?>
<![CDATA[ a > 5 & b < 10 ]]>
<?GS print(format = TRUE)>
<!DOCTYPE Movie>
<!-- This is a commet -->
<movie mins="126" lang="en">
  <title>Good Will Hunting</title>
  <director>
    <first_name>Gus</first_name>
    <last_name>Van Sant</last_name>
  </director>
  <year>1998</year>
  <genre>drama</genre>
</movie>
```

The following table lists some of the common additional XML elements:

Markup	Name	Description
`<?xml >`	XML Declaration	Identifies content as an XML document
`<?PI >`	Processing Instruction	Processing instructions passed to application PI
`<!DOCTYPE >`	Document-type Declaration	Defines the structure of an XML document
`<![CDATA[ ]]>`	CDATA Character Data	Anything inside a CDATA is ignored by the parser
`<!--  -->`	Comment	For writing comments

4.1.5 Another Example

Let's go back to the *movie* example, but now let's see how the content of our hypothetical XML document should look like:

```
<?xml version="1.0"?>
<!DOCTYPE movies>
<movie mins="126" lang="en">
  <!-- this is a comment -->
  <title>Good Will Hunting</title>
  <director>
    <first_name>Gus</first_name>
```

```
    <last_name>Van Sant</last_name>
  </director>
  <year>1998</year>
  <genre>drama</genre>
</movie>
```

Each Node can have

- a Name
- any number of attributes
- optional content
- other nested elements

4.1.6 Wrapping-Up

About XML

- designed to store and transfer data
- designed to be self-descriptive
- tags are not predefined and can be extended
- a generic language that provides structure and syntax for many markup dialects
- is a syntax or format for defining markup languages
- a standard for the semantic, hierarchical representation of data
- provides a general approach for representing all types of information dialects

Chapter 5

Parsing XML

The goal of this chapter is to describe how we can parse XML content with the R package `xml2`

You will need the following packages

```
library(xml2)
library(stringr)
```

We'll cover a variety of situations you most likely will find yourself dealing with:

- R package `"xml2"`
- Navigating the XML tree structure
- XPath expressions

5.1 What is parsing?

Getting data from the web often involves reading and processing content from XML and HTML documents. This is known as **parsing**.

The dictionary defines "parse" as:

> analyze (a sentence) into its parts and describe their syntactic roles.

In regards to "computing", parse has to do with:

> analyze (a string or text) into logical syntactic components, typically in order to test conformability to a logical grammar.
>
> an act of or the result obtained by parsing a string or a text.

According to Wikipedia, a parser is:

> A parser is a software component that takes input data (frequently text) and builds a data structure —often some kind of parse tree, abstract syntax tree or other hierarchical structure— giving a structural representation of the input, checking for correct syntax in the process

5.2 R package "xml2"

The package "xml2" is designed for one major purpose, namely, to parse XML and HTML content. Remember that HTML is one the countless XML dialects.

As of this writing, "xml2" has minimal functionality for writing content in XML. Hadley Wickham has mentioned that he plans to add more functions for writing XML. So it is possible that in the future, "xml2" integrates more writing-XML functionality. Having said that, we will focus exclusively on reading XML content.

We'll cover 4 major types of tasks that we can perform with "xml2"

- parsing (ie *reading*) xml / html content
- obtaining descriptive information about parsed contents
- navigating the tree structure (i.e. accessing its components)
- querying and extracting data from parsed contents

5.2.1 Parsing Functions

There are two main parsing functions:

- `read_xml()`
- `read_html()`

For XML files in general, you should use `read_xml()`. For HTML files, then it's better to use `read_html()` because it is more robust, and can handle no well-formed HTML files, which are not uncommon to deal with in practice.

The main input for these reading functions is either a string, an R connection, or a raw vector.

The string can be either a path, a URL or literal xml. URL's will be converted into connections either using `base::url()` or, if installed, `curl::curl()`. Local paths ending in .gz, .bz2, .xz, .zip will be automatically uncompressed.

Both `read_xml()` and `read_html()` return an object of class `"xml_document"`.

Let's see an example. Consider one of the examples from the previous chapter, for instance some content in XML:

```
<movie mins="126" lang="en">
  <title>Good Will Hunting</title>
  <director>
```

```
    <first_name>Gus</first_name>
    <last_name>Van Sant</last_name>
  </director>
  <year>1998</year>
  <genre>drama</genre>
</movie>
```

For illustration purposes, let's take the XML content, treating it as a single character string, that we then pass to `read_xml()`:

```
# toy example with xml string
movie <- read_xml(
"<movie>
<title>Good Will Hunting</title>
<director>
<first_name>Gus</first_name>
<last_name>Van Sant</last_name>
</director>
<year>1998</year>
<genre>drama</genre>
</movie>")

movie
#> {xml_document}
#> <movie>
#> [1] <title>Good Will Hunting</title>
#> [2] <director>\n  <first_name>Gus</first_name>\n  <last_name>Van Sant</last_n ...
#> [3] <year>1998</year>
#> [4] <genre>drama</genre>
```

As we mention, the `movie` is an XML object:

```
class(movie)
#> [1] "xml_document" "xml_node"
```

This type of object has an internal structure in order to maintain the hierarchical tree-structure of any XML content.

5.3 Working with parsed documents

Having parsed an XML / HTML document, we can use 2 main functions to start working on the tree structure:

- `xml_root()` gets access to the root node and its elements
- `xml_children()` gets access to the children nodes of a given node

5.3.1 Example with a basic XML document

Here's some content: a movie elements in XML syntax

xml string movie

```
<movie>
   <title>
      Good Will Hunting
   </title>
   <director>
      <first_name>Gus</first_name>
      <last_name>Van Sant</last_name>
   </director>
   <year>1998</year>
   <genre>drama</genre>
</movie>
```

Figure 5.1: XML Movie

The following figure identifies the main nodes:

Below is an abstract representation of an XML file, and its main nodes

5.3.2 More Functions in "xml2"

In addition to `xml_root()` and `xml_children()`, there are other functions to parse the various kinds of content within a given node.

Here's a table with the main navigation functions. Keep in mind that the applicability of the functions depends on the class of objects we are working on.

Function	Description
`xml_root()`	Returns root node
`xml_children()`	Returns children nodes
`xml_child()`	Returns specified children number
`xml_name()`	Returns name of a node
`xml_contents()`	Returns contents of a node
`xml_text()`	Returns text
`xml_length()`	Returns number of children nodes
`xml_parents()`	Returns set of parent nodes
`xml_siblings()`	Returns set of sibling nodes

Figure 5.2: XML Movie nodes

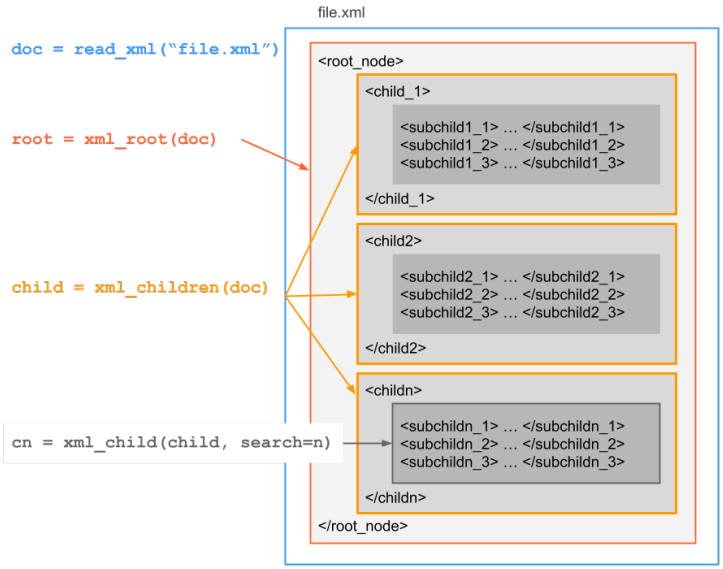


Figure 5.3: Functions of ‘xml2‘

5.3.3 Navigation of XML / HTML Tree

Let's consider the following XML content:

```
<movies>
     <movie mins="126" lang="eng">
        <title>Good Will Hunting</title>
        <director>
           <first_name>Gus</first_name>
           <last_name>Van Sant</last_name>
        </director>
        <year>1998</year>
        <genre>drama</genre>
     </movie>
     <movie mins="106" lang="spa">
        <title>Y tu mama tambien</title>
        <director>
           <first_name>Alfonso</first_name>
           <last_name>Cuaron</last_name>
        </director>
        <year>2001</year>
        <genre>drama</genre>
     </movie>
  </movies>
```

Theis content can be depicted in the following tree-diagram:

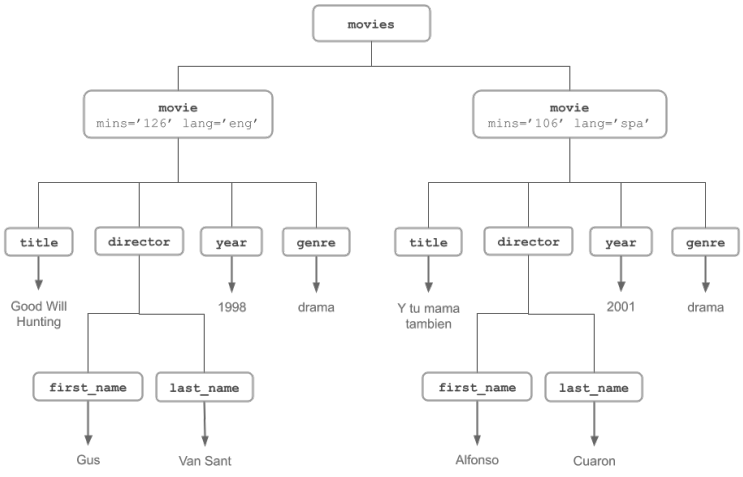


Figure 5.4: XML movies tree

Let's create a character vector to store the XML content:

```
# toy example with xml string
xml_string <- c(
  '<?xml version="1.0" encoding="UTF-8"?>',
  '<movies>',
  '<movie mins="126" lang="eng">',
  '<title>Good Will Hunting</title>',
  '<director>',
  '<first_name>Gus</first_name>',
  '<last_name>Van Sant</last_name>',
  '</director>',
  '<year>1998</year>',
  '<genre>drama</genre>',
  '</movie>',
  '<movie mins="106" lang="spa">',
  '<title>Y tu mama tambien</title>',
  '<director>',
  '<first_name>Alfonso</first_name>',
  '<last_name>Cuaron</last_name>',
  '</director>',
  '<year>2001</year>',
  '<genre>drama</genre>',
  '</movie>',
  '</movies>')
```

Let's parse the content. To do this, we must first create a single contiguous xml string, which is done with `paste()` and its `collapse = ''` argument:

```
# parsing xml string
doc <- read_xml(paste(xml_string, collapse = ''))

doc
#> {xml_document}
#> <movies>
#> [1] <movie mins="126" lang="eng">\n  <title>Good Will Hunting</title>\n  <dir ...
#> [2] <movie mins="106" lang="spa">\n  <title>Y tu mama tambien</title>\n  <dir ...
```

And let's navigate the tree structure. We begin with `xml_root()` to get access to the root node:

```
# root node
movies <- xml_root(doc)
movies
#> {xml_document}
#> <movies>
#> [1] <movie mins="126" lang="eng">\n  <title>Good Will Hunting</title>\n  <dir ...
#> [2] <movie mins="106" lang="spa">\n  <title>Y tu mama tambien</title>\n  <dir ...
```

It turns out that `doc` and `movies` are actually identical:

```
identical(doc, movies)
#> [1] TRUE
```

We use the `xml_length()` to know how many elements or nodes are in the root node:

```
# parsing xml string
xml_length(doc)
#> [1] 2
```

which confirms what we know about the movies string that contains two `movie` elements: one node for "Good Will Hunting" and another node for "Y tu mama tambien".

The function `xml_children()` allows you to access the children nodes:

```
xml_children(doc)
#> {xml_nodeset (2)}
#> [1] <movie mins="126" lang="eng">\n  <title>Good Will Hunting</title>\n  <dir ...
#> [2] <movie mins="106" lang="spa">\n  <title>Y tu mama tambien</title>\n  <dir ...
```

Notice that the output is an object of class `"xml_nodeset"`. To access a specific node, you use the function `xml_child()`. In this example, the node for movie "Good Will Hunting" corresponds to the first node, and we pass this value to the `search` argument:

```
xml_child(doc, search = 1)
#> {xml_node}
#> <movie mins="126" lang="eng">
#> [1] <title>Good Will Hunting</title>
#> [2] <director>\n  <first_name>Gus</first_name>\n  <last_name>Van Sant</last_n ...
#> [3] <year>1998</year>
#> [4] <genre>drama</genre>
```

Likewise, the second node ("Y tu mama tambien") is accessed by specifying the argument `search = 2`:

```
xml_child(doc, search = 2)
#> {xml_node}
#> <movie mins="106" lang="spa">
#> [1] <title>Y tu mama tambien</title>
#> [2] <director>\n  <first_name>Alfonso</first_name>\n  <last_name>Cuaron</last ...
#> [3] <year>2001</year>
#> [4] <genre>drama</genre>
```

This is the view of the tree structure so far:

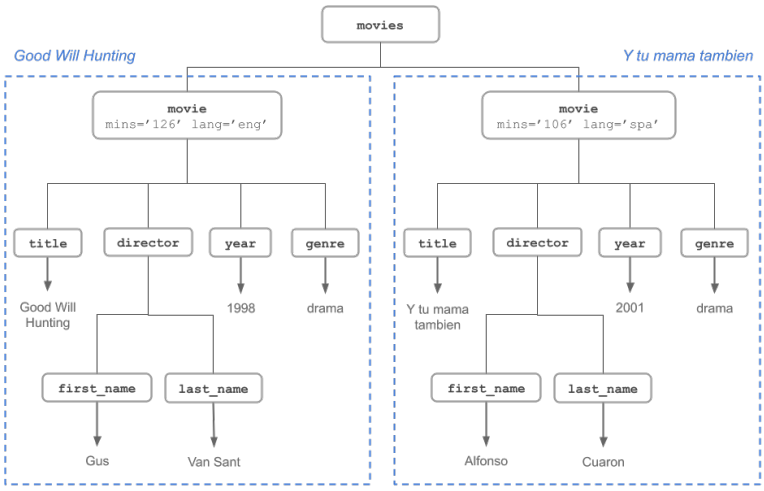


Figure 5.5: Two movie nodes

Inspecting first node

Let's go inside the first node, and store the content in the object `good_will`

```
# first child
good_will <- xml_child(doc, search = 1)
good_will
#> {xml_node}
#> <movie mins="126" lang="eng">
#> [1] <title>Good Will Hunting</title>
#> [2] <director>\n  <first_name>Gus</first_name>\n  <last_name>Van Sant</last_n ...
#> [3] <year>1998</year>
#> [4] <genre>drama</genre>
```

and let's do the same for the second node, storing the content in the object `tu_mama`:

```
# second child
tu_mama <- xml_child(doc, search = 2)
tu_mama
#> {xml_node}
#> <movie mins="106" lang="spa">
#> [1] <title>Y tu mama tambien</title>
#> [2] <director>\n  <first_name>Alfonso</first_name>\n  <last_name>Cuaron</last ...
#> [3] <year>2001</year>
#> [4] <genre>drama</genre>
```

We can then again apply `xml_children()` on each node to see what children

nodes `good_will` and `tu_mama` have:

```
# children of good_will
xml_children(good_will)
#> {xml_nodeset (4)}
#> [1] <title>Good Will Hunting</title>
#> [2] <director>\n  <first_name>Gus</first_name>\n  <last_name>Van Sant</last_n ...
#> [3] <year>1998</year>
#> [4] <genre>drama</genre>
```

```
# children of tu_mama
xml_children(tu_mama)
#> {xml_nodeset (4)}
#> [1] <title>Y tu mama tambien</title>
#> [2] <director>\n  <first_name>Alfonso</first_name>\n  <last_name>Cuaron</last ...
#> [3] <year>2001</year>
#> [4] <genre>drama</genre>
```

The visual diagram for `good_will` depicts the four nodes:

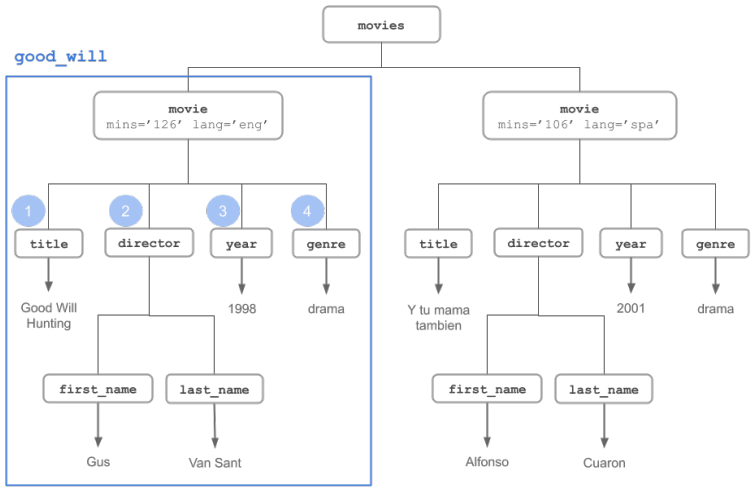


Figure 5.6: Children nodes of 'Good Will Hunting'

The code below shows a deeper inspection of `good_will`. The function `xml_name()` gives the name of a node.

```
# name of an element
xml_name(good_will)
#> [1] "movie"
```

The function `xml_attrs()` gives you the attributes of a node. In this case, the node of `good_will` has to attributes `"mins"` and `"lang"`

```
# attributes
xml_attrs(good_will)
#>  mins  lang
#> "126" "eng"
```

As we previously saw, `xml_length()` gives the number of children nodes inside a given node:

```
# how many children
xml_length(good_will)
#> [1] 4
```

Likewise, we can move along the children nodes, and find information about their names, their subchildren, and so on:

```
# name of children (of good_will)
xml_name(xml_children(good_will))
#> [1] "title"    "director" "year"     "genre"

# good_will title
xml_child(good_will, "title")
#> {xml_node}
#> <title>
```

```
# good_will title
title1 <- xml_child(good_will, "title")
title1
#> {xml_node}
#> <title>

# content good_will title
xml_contents(title1)
#> {xml_nodeset (1)}
#> [1] Good Will Hunting

# text good_will title
xml_text(title1)
#> [1] "Good Will Hunting"
```

5.3.3.1 Inspecting director node

```
# good_will director
dir1 <- xml_child(good_will, "director")
dir1
#> {xml_node}
#> <director>
#> [1] <first_name>Gus</first_name>
```

```
#> [2] <last_name>Van Sant</last_name>
```

```
xml_children(dir1)
#> {xml_nodeset (2)}
#> [1] <first_name>Gus</first_name>
#> [2] <last_name>Van Sant</last_name>
```

To extract just the text, we use `xml_text()`:

```
xml_text(dir1)
#> [1] "GusVan Sant"
```

The visual diagram for `good_will` with its director node is depicted in this figure:

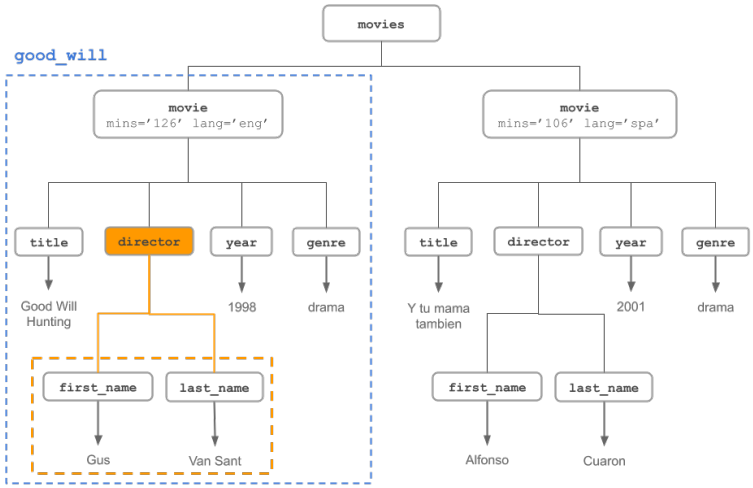


Figure 5.7: Director node of 'Good Will Hunting'

Chapter 6

XPath Language

In the preceding chapter you learned about the main functions in `"xml"` that allows us to parse XML and HTML documents. While those functions can be quite useful to navigate through the elements of an XML document, their default usage can be a bit limiting.

The real parsing power comes from the ability to **locate nodes and extract information from them**. For this, we need to be able to perform queries on the parsed content. The solution is provided by **XPath**, which is a language to navigate through elements and attributes in an XML/HTML document.

6.1 What is XPath?

XPath is a language for finding information in an XML document. It works by identifying patterns to match data or content. To be more precise, XPath uses **path expressions** to select nodes in an XML document by taking into account the tree structure of XML based on:

- node names
- node content
- a node's relationship to other nodes

6.1.1 XPath Syntax

The key concept is knowing how to write XPath expressions. XPath expressions have a syntax similar to the way files are located in a hierarchy of directories and folders in a computer file system. For instance:

```
/movies/movie
```

is the XPath expression to locate the `movie` children in the `movies` (root) element

6.1.2 Selecting Nodes

The main symbols to define path expressions are:

Symbol	Description
`/`	selects from the root node
`//`	selects nodes anywhere
`.`	selects the current node
`..`	selects the parent of the current node
`@`	selects attributes
`[]`	square brackets to indicate attributes
`*`	matches any element node
`@*`	matches any attribute node

For instance:

Example	Description
`/node`	selects top level node
`//node`	selects nodes at any level
`node[@attr]`	node that has an attribute named `attr`
`node[@attr="abc"]`	node that has an attribute named `attr` with value `"abc"`
`node/@attr`	value of an attribute `attr` in node with such attribute
`node/*`	any (child) element in node
`node/@*`	value of any attribute in node

6.2 XPath Examples

To make things less abstract, let's bring back the `movies` XML document containing two movies *Good Will Hunting* and *Y tu mama tambien*

The following diagrams illustrate different XPath expressions to match and select nodes based on either: their names, their content, or their relationship to other nodes.

6.3 Using XPath Functions

The R package `"xml2"` provides a large number of functions that admit XPath expressions; these functions have the `xpath` argument. The following code snip-

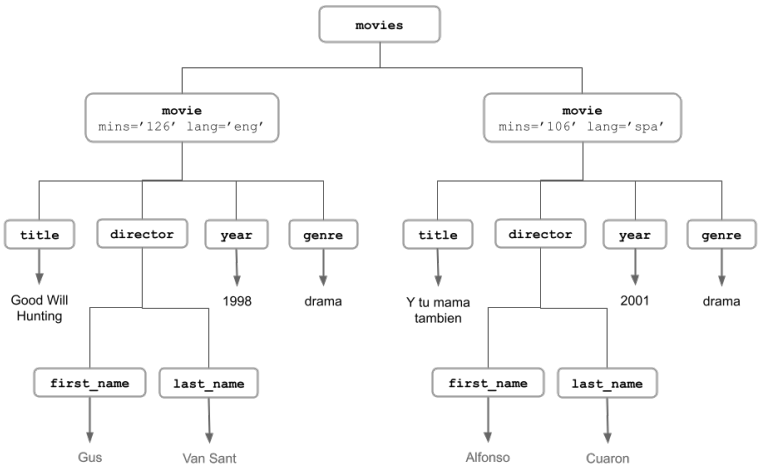


Figure 6.1: XML movies

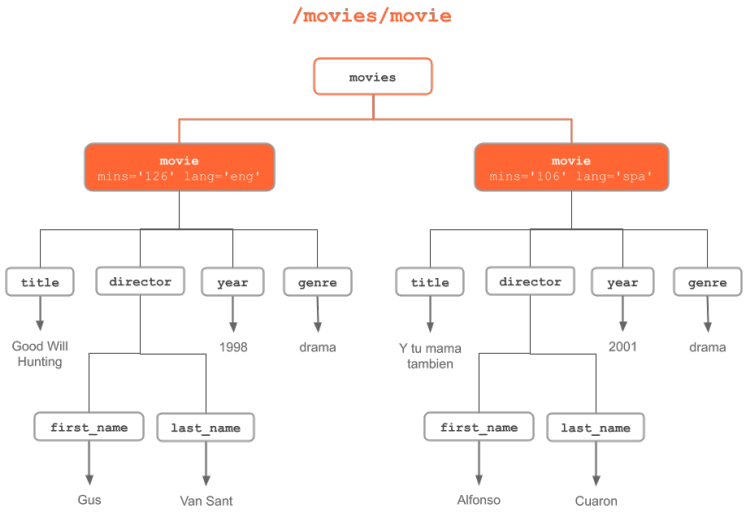


Figure 6.2: "movie" nodes

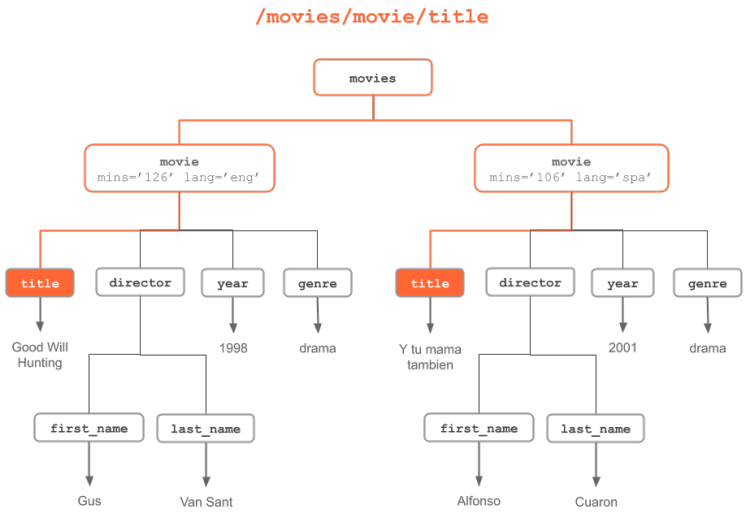


Figure 6.3: "title" nodes

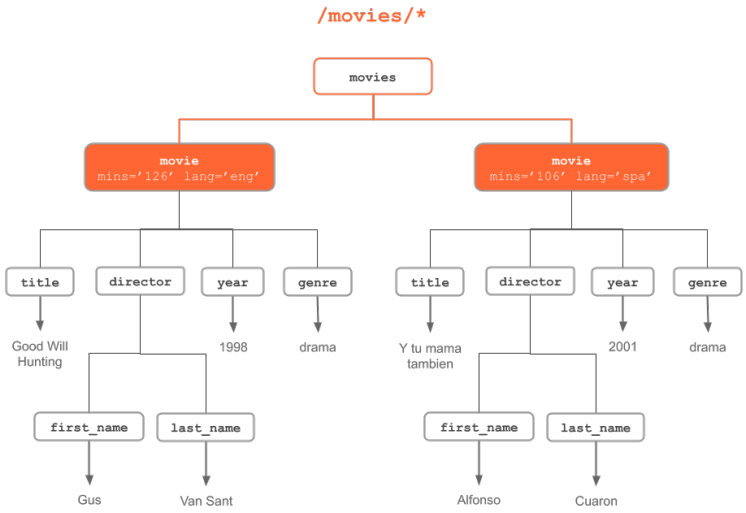


Figure 6.4: Any nodes of "movies" node

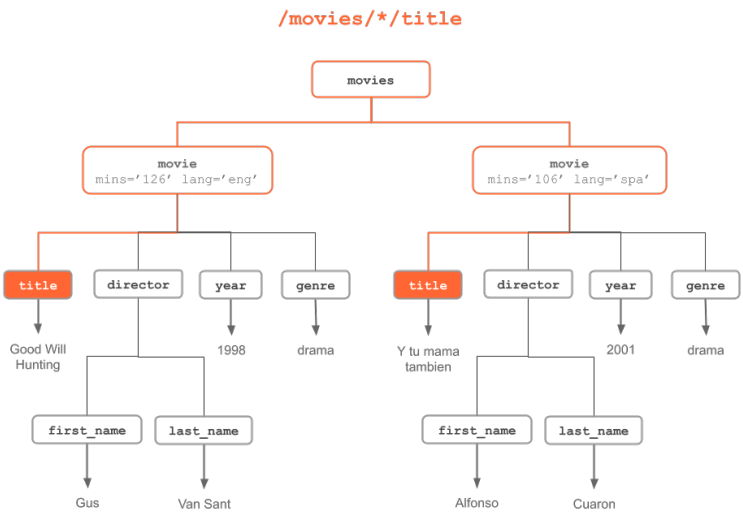


Figure 6.5: Another way to select "title" nodes

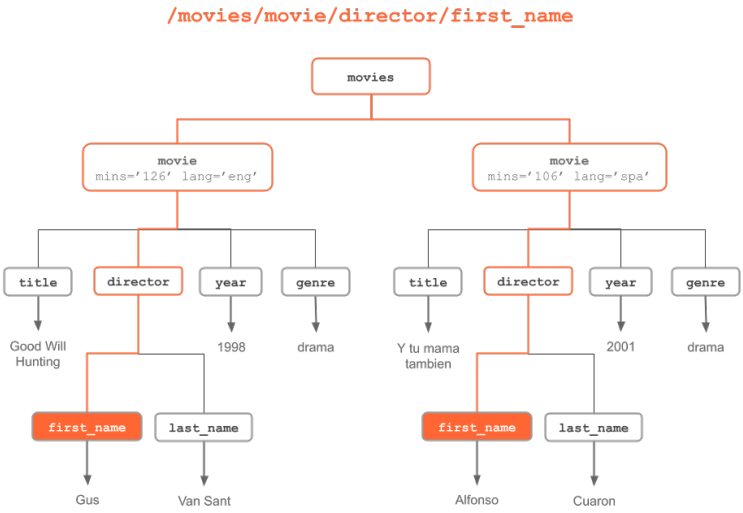


Figure 6.6: "first name" nodes

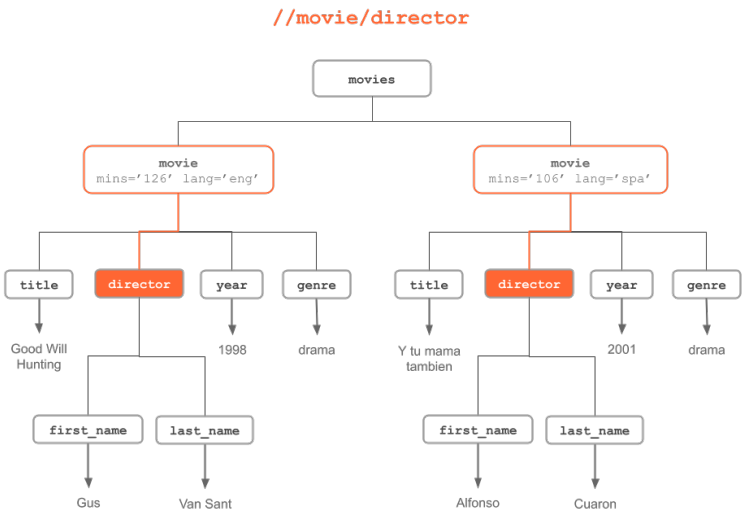


Figure 6.7: "movie/director" anywhere in the XML tree

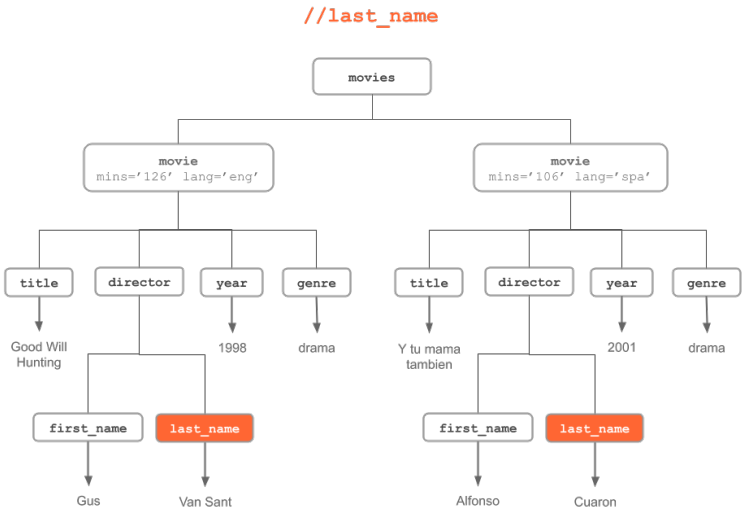


Figure 6.8: "last name" nodes anywhere in the XML tree

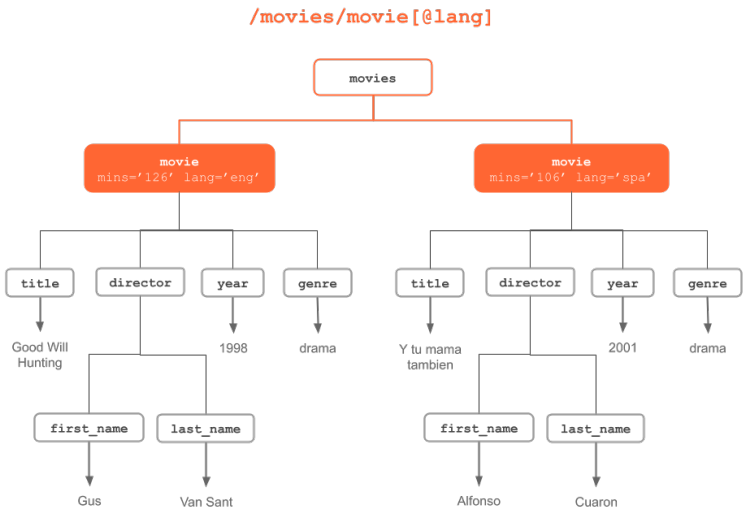


Figure 6.9: "movie" nodes having "lang" attribute

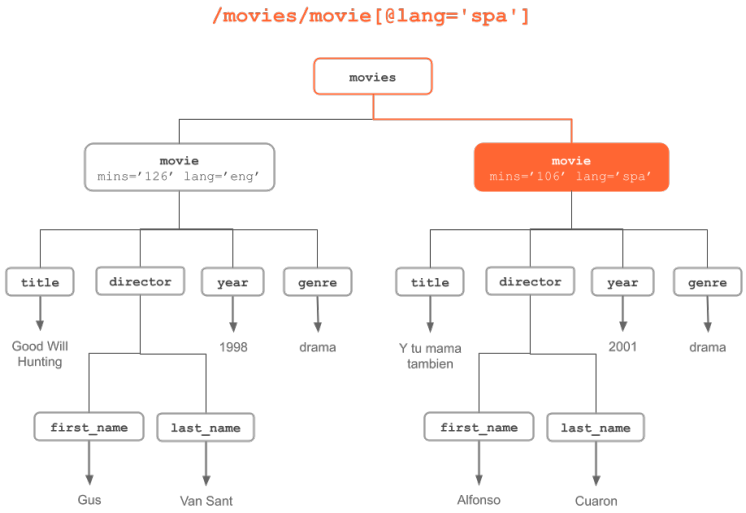


Figure 6.10: "movie" nodes with "lang" attribute having value "spa"

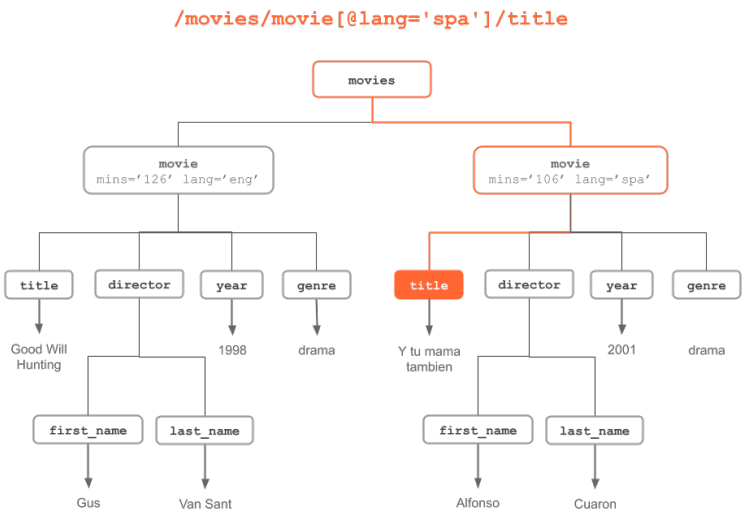


Figure 6.11: "title" node of movie with spanish language attribute

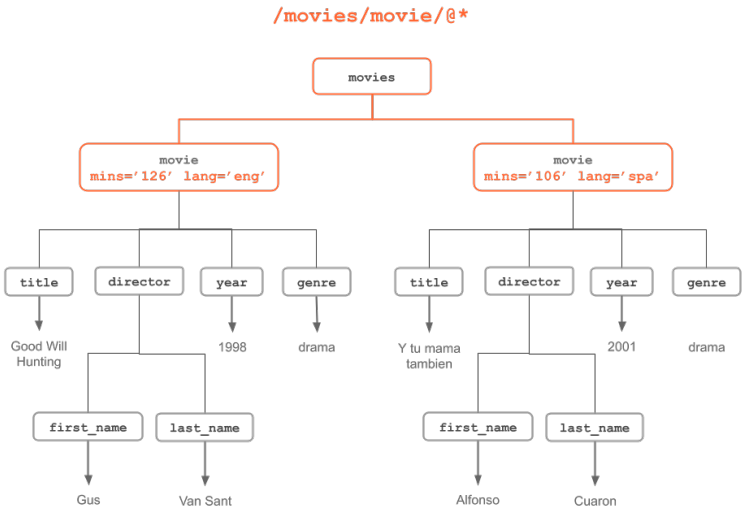


Figure 6.12: Any atrribute of "movie" nodes

pets, based on the above pattern examples, show you how to use some of these functions.

```
# toy example with xml string
xml_string <- c(
  '<?xml version="1.0" encoding="UTF-8"?>',
  '<movies>',
  '<movie mins="126" lang="eng">',
  '<title>Good Will Hunting</title>',
  '<director>',
  '<first_name>Gus</first_name>',
  '<last_name>Van Sant</last_name>',
  '</director>',
  '<year>1998</year>',
  '<genre>drama</genre>',
  '</movie>',
  '<movie mins="106" lang="spa">',
  '<title>Y tu mama tambien</title>',
  '<director>',
  '<first_name>Alfonso</first_name>',
  '<last_name>Cuaron</last_name>',
  '</director>',
  '<year>2001</year>',
  '<genre>drama</genre>',
  '</movie>',
  '</movies>')

# parsing xml string
doc = read_xml(paste(xml_string, collapse = ''))
```

```
# movie children (from root node)
movie_nodes = xml_find_all(doc, xpath = "/movies/movie")
movie_nodes
#> {xml_nodeset (2)}
#> [1] <movie mins="126" lang="eng">\n  <title>Good Will Hunting</title>\n  <dir ...
#> [2] <movie mins="106" lang="spa">\n  <title>Y tu mama tambien</title>\n  <dir ...
```

```
# title children (from root node)
title_nodes = xml_find_all(doc, xpath = "/movies/movie/title")
title_nodes
#> {xml_nodeset (2)}
#> [1] <title>Good Will Hunting</title>
#> [2] <title>Y tu mama tambien</title>
```

```
# text content of title_nodes
xml_text(title_nodes)
#> [1] "Good Will Hunting" "Y tu mama tambien"
```

```
# director children (from any movie element)
director_nodes = xml_find_all(doc, "//movie/director")
director_nodes
#> {xml_nodeset (2)}
#> [1] <director>\n  <first_name>Gus</first_name>\n  <last_name>Van Sant</last_n ...
#> [2] <director>\n  <first_name>Alfonso</first_name>\n  <last_name>Cuaron</last ...
```

```
# text content of director_nodes
xml_text(director_nodes)
#> [1] "GusVan Sant"   "AlfonsoCuaron"
```

```
# last_name (from anywhere in the tree)
last_name_nodes = xml_find_all(doc, "//last_name")
last_name_nodes
#> {xml_nodeset (2)}
#> [1] <last_name>Van Sant</last_name>
#> [2] <last_name>Cuaron</last_name>
```

```
# text of last_name (from anywhere in the tree)
xml_text(last_name_nodes)
#> [1] "Van Sant" "Cuaron"
```

```
# title node of movie with attribute lang='spa'
title_spa = xml_find_all(doc, "/movies/movie[@lang='spa']/title")
title_spa
#> {xml_nodeset (1)}
#> [1] <title>Y tu mama tambien</title>
```

```
# text content of title_spa
xml_text(title_spa)
#> [1] "Y tu mama tambien"
```

Part III

HTML

Chapter 7

Basics of HTML

The goal of this chapter is to give you a crash introduction to HTML, so you can get a good grasp of this format before moving to the next chapter.

7.1 A quick introduction to HTML

HTML is not a programming language; it is simply a markup language, which means it is a syntax for identifying and describing the elements of a document such as headings, paragraphs, lists, tables, images, hyperlinks, etc. Technically, HTML is an XML dialect.

Say we visit R's official website (screencapture below).

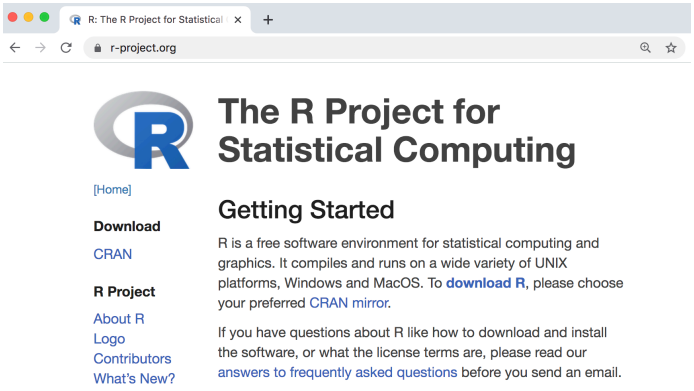


Figure 7.1: R project's home page

The visually rich and interactive pages we see on the Web are based on plain text files referred to as *source* files. To look at the actual HTML content behind

R's homepage, you need to get access to the source code option in your browser. If you are using Chrome, go to the **View** tab in the menu bar, then choose the **Developer** option, and finally click on **View Source**.

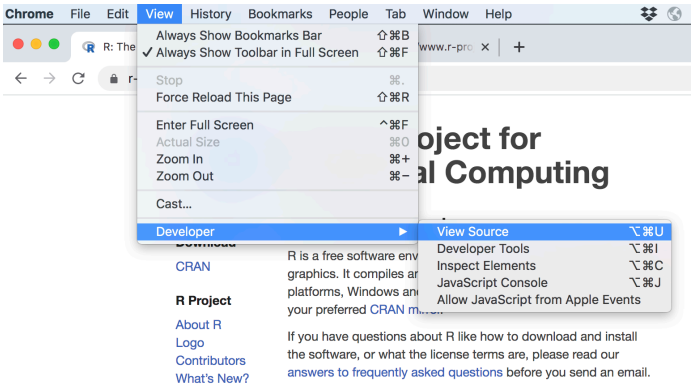


Figure 7.2: View source code of a webpage in Chrome

If we take a look at the source file behind R's homepage, we'll discover the actual HTML content, depicted in the image below.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>R: The R Project for Statistical Computing</title>

    <link rel="icon" type="image/png" href="/favicon-32x32.png" sizes="32x32" />
    <link rel="icon" type="image/png" href="/favicon-16x16.png" sizes="16x16" />

    <!-- Bootstrap -->
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/R.css" rel="stylesheet">

    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->
    <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    <div class="container page">
      <div class="row">
        <div class="col-xs-12 col-sm-offset-1 col-sm-2 sidebar" role="navigation">
<div class="row">
<div class="col-xs-6 col-sm-12">
<p><a href="/"><img src="/Rlogo.png" width="100" height="78" alt = "R" /></a></p>
<p><small><a href="/">[Home]</a></small></p>
<h2 id="download">Download</h2>
<p><a href="http://cran.r-project.org/mirrors.html">CRAN</a></p>
<h2 id="r-project">R Project</h2>
```

Figure 7.3: HTML source code behind R project's home page

As you can tell, the webpage is cleverly rendered by your browser that knows exactly how to take care of the content in the source file. If you are not familiar with HTML, some (if not most) of the text will look like gibberish to you right now. But it all has a specific structure and meaning.

What you see on the browser is the result of the resources served by the server

where R's website is stored. Technically speaking, the resources should include an `index.html`file, plus other files (stylesheet files, and image files)

Figure 7.4: Other resources being linked in the home page

In particular, the following resources (different types of files) can be identified:

- `index.html`
- `favicon-33x32.png`
- `favicon-16x16.png`
- `bootstrap.min.css`
- `R.css`
- `Rlogo.png`

7.1.1 HTML document structure

Let's study the structure of a basic HTML document. Below is a diagram with a simplified content of R's webpage.

The first line of text is the *document type declaration*, which identifies this document as an HTML5 document. Then we have the **html** element which is the root element of the document, and it contains all the other elements.

Within the **html** element, we find two elements: the **head** and the **body**. The *head* element contains descriptive information such as the title, style sheets, scripts, and other meta information. The mandatory element inside the head is the **title**.

The **body** element contains everything that is displayed in the browser.

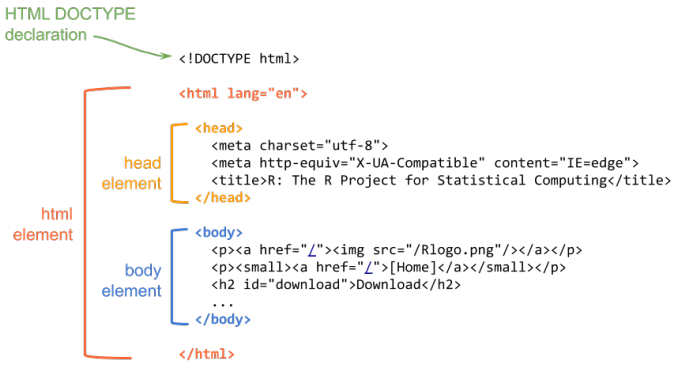


Figure 7.5: HTML document structure

7.1.2 HTML Syntax

You don't need to memorize all possible HTML elements (or tags), but it's important that you learn about their syntax and structure. So let's describe the anatomy of html elements.

Here's an example with a `<p>` element which is the **paragraph** element. An HTML tag has an opening tag consisting of the tag name surrounded by angle brackets, that is, the `<p>` characters.

Usually, you put tags around some *content* text. At the end of the tag there is the closing tag, in this case `</p>`. You know it's a closing tag because it comes after the content, and it has a slash `/` before the `p` name. All closing tags have a slash in them.

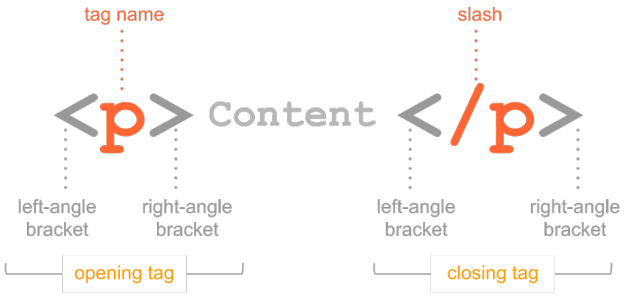


Figure 7.6: Anatomy of html elements

Not all tags come in the form of a pair of matching tags (an opening and a closing tag). There are some tags that don't have a closing tag. Perhaps the most common tag of this type is the `<img>` tag used for images. One example

is the `<img>` tag for the R logo file in the homepage of R project:

```
<img src="/Rlogo.png"/>
```

As you can tell, the `<img>` tag does not have a closing tag; you can say that itself closes with a slash and the right angle bracket `/>`.

Some elements have **attributes** which allows you to specify additional information about an element. Attributes are declared inside the opening tag using special keywords. We assign values to attributes with the equals sign, and we specify the values inside quotations.

Figure 7.7: Attributes and values in html tags

In the example above, a paragraph tag contains an attribute `lang` for *language* with a value of `es` for *español* or spanish.

Notice also that the previous `<img>` element has an attribute `src` to indicate the source filename of the picture, in this case, `"/Rlogo.png"`.

7.1.3 What the browser does

The browser (e.g. Chrome, Safari, Firefox) reads the HTML, interprets all the tags, and renders the content accordingly. Recall that tags tell browser about the structure and meaning of the text. The browser identifies what parts are headings (e.g. `<h1>`, `<h2>`), what parts are paragraphs (e.g. `<p>`), what parts are lists (e.g. `<ol>`, `<ul>`), what text needs to be emphasized, and so on.

The HTML syntax tells the browser about the structure of a document: where the headings are, where the paragraphs are, what text is part of a list, and so on. How do browsers know this? Well, they have built-in default rules for how to render HTML elements. In addition to the default settings, HTML elements can be formatted in endless ways using what is called Cascade Style Sheets or CSS for short, that determine font types, colors, sizes, and many other visual aspects of a page.

7.1.4 Web Scraping

Many websites are secured by an SSL/TSL certificate, which you can identify by looking at the URL containing `https` (Hyper Text Transfer Protocol Secure). SSL stands for **Secure Sockets Layer**. This is a technology that keeps an internet connection secure and safeguards sensitive data that is being sent between a

client and a server (for example, when you use your browser to shop in amazon) or server to server (for example, an application with payroll information). The SSL technology is currently deprecated and has been replaced entirely by TLS which stands for **Transport Layer Security**. Simply put, TSL also ensures data privacy the same way that SSL does. Since SSL is actually no longer used, this is the correct term that people should start using.

HTTPS is a secure extension of HTTP. When a website uses HTTPS it means that the website is secured by an SSL/TLS certificate. Consequently, websites that install and configure an SSL/TLS certificate can use the HTTPS protocol to establish a secure connection with the server. **Quote**: "The details of the certificate, including the issuing authority and the corporate name of the website owner, can be viewed by clicking on the lock symbol on the browser bar."

Wikipedia uses HTTPS. For instance, if we visit the entry for men's long jump world record progression, the url is

https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progressi
on

If we try to use functions like `readHTMLTable` from `"XML"` package, it will fail

```
wiki <- 'https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression'

# this fails
tbls <- readHTMLTable(wiki)
```

One option to read the html tables and extract them as R data frames, is to first download the html file to your computer, and then use `readHTMLTable()` to scrape the tables:

```
# desired url
wiki <- 'https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression'

# destination file
jump_html <- 'men-long-jump-records.html'

# download file to your working directory
download.file(wiki, jump_html)

tbls <- readHTMLTable(jump_html)
```

We recommend using this option when:

- the data fits in your computer, in this way you also have the *raw data*
- you need to experiment and get to know the content, in order to decide which elements you will extract, which functions to use, what kind of processing operations or transformations you need to apply, etc.
- also, downloading an HTML document save you from making innecessary requests that could get in trouble, and potentially be blocked by a server

because you are overloading them with multiple requests.

Part IV

JSON

Chapter 8

JSON Data

The goal of this chapter is to provide an introduction for handling JSON data in R.

We'll cover the following topics:

- JSON Basics
- R packages for JSON data
- Reading JSON data from the Web

8.1 JSON Basics

JSON stands for **JavaScript Object Notation** and it is a format for representing data. More formally, we can say that it is a text-based way to store and transmit structured data. By using a simple syntax, you can easily store anything from a single number to strings, JSON-arrays, and JSON-objects using nothing but a string of plain text. As you will see, you can also nest arrays and objects, allowing you to create complex data structures.

8.2 What is JSON?

Let's first talk about what JSON is and why it is important.

JSON is a data representation format very similar to XML. It's used widely across the internet for almost every single API that you will access as well as for *config* files and things such as games and text editors. Its popularity is based on a handful of attractive aspects:

- It's extremely lightweight and compact to send back and forth due to the small size file;

- It's easy for both computers and people to read-and-write, compared to something like XML, since it's much cleaner and there's not as many opening and closing tags;
- It maps very easily onto the data structures used by most programming languages (numbers, strings, booleans, nulls, arrays and associative arrays);
- It also integrates very nicely with javascript since JSON is just a superset of javascript which means anything you write in JSON is valid javascript, which is a language used all throughout the web for front-end or back-end of applications.
- Also, every single major language has some form of library or packages with built-in functionality to parse JSON strings into objects or classes in that language which makes working with JSON data extremely easy inside of a programming language.

Why should we care about JSON? When working with data from the Web, we'll inevitably find some JSON data because it is commonly used in web applications to send data from the server to the browser. As a matter of fact, in your data science career you will be using JSON quite often, whether it is consuming an API, creating an API, or creating *config* files for you or other people to use for your application.

8.3 Understanding JSON Syntax

Let's now talk about the syntax used to store and organize data in JSON.

8.3.1 Data Types

The first thing to talk about is the **data types** or **values** that JSON can represent. As we know, JSON is a data representation format, so we need to be able to represent certain data types within it. JSON supports the following types:

- `string` (in double quotes)
- `number` (in any format whether they're decimal numbers, integers, negative numbers, even numbers in scientific notation)
- `true` and `false` (booleans)
- `null`

8.3.2 Arrays

JSON also supports **arrays** (in JSON Sense) which are sets of data types defined within brackets, and contains a comma-separated list of values. For example

`[1, 3, 3]` or `["computing", "with", "data"]`, which can be a set of any of the data types listed above.

We typically use arrays when we have a set of unnamed values, this is why some people refer to them as **ordered unnamed arrays**. The closest R object to a JSON array would be a vector:

- JSON: `[1, 2, 3, ... ]`; -vs- R: `c(1, 2, 3, ...)`
- JSON: `[true, true, false, ... ]`; -vs- R: `c(TRUE, TRUE, FALSE, ...)`

8.3.3 Objects

Another type of data container is the so-called JSON **object**, which is the most complex but also the most used type of object within JSON, and it allows you to represent values that are key-value pairs:

```
{"key": "value"}
```

You use curly braces to define a JSON-object, and inside the braces you put key-value pairs. The key must be surrounded by double quotes, followed by a colon, followed by the value. The value can be a single data type, but it can also be a JSON-array (which in turn can contain a JSON-object). Because you have the association of a *key* with its *value*, these JSON structures are also referred to as associative arrays.

For example, say the key is `"year"` and the value `2000`, then a simple JSON object will look like this:

```
{"year": 2000}
```

Another example can be a key `"name"` and a value `"Jessica"`:

```
{"name": "Jessica"}
```

If you have multiple key-value pairs, you separate each of them with a comma:

```
{
  "name1": "Nicole",
  "name2": "Pleuni",
  "name3": "Rori"
}
```

A more complex object might look like the following example. In this case we have JSON-object that contains three key-value pairs. Each of the keys is a `"person"` and the associated pair corresponds to an array which in turn contains a JSON-object with two key-value pairs: the *first name*, and the *last name*:

```
{
  "person1": [
    {
```

```
      "first": "Nicole",
      "last": "Adelstein"
    }
  ],
  "person2": [
    {
      "first": "Pleuni",
      "last": "Pennings"
    }
  ],
  "person3": [
    {
      "first": "Rori",
      "last": "Rohlfs"
    }
  ]
}
```

Because the data inside a JSON object is formed of key-value pairs, you could think of them as **named arrays**.

What do JSON-objects correspond to in R? Well, there's not really a unique correspondence between a JSON-object and its equivalent in structure R. For instance, let's bring back one of the JSON-objects previously discussed:

```
{
  "name1": "Nicole",
  "name2": "Pleuni",
  "name3": "Rori"
}
```

We could use a named R vector to store the same data:

```
# named vector in R
c("name1" = "Nicole", "name2" = "Pleuni", "name3" = "Rori")
```

But we could also use an R list:

```
# named list in R
list("name1" = "Nicole", "name2" = "Pleuni", "name3" = "Rori")
```

Keep in mind that JSON-objects can be more complex than this basic example. Because JSON objects can contain any other type of JSON data structure in them, the similar container in R to a JSON-object is a `list`.

8.3.4 Examples of JSON Data Containers

Here's a series of examples involving combinations of JSON arrays and objects.

JSON containers can be nested. Here's one example:

```
{
    "name": ["X", "Y", "Z"],
    "grams": [300, 200, 500],
    "qty": [4, 5, null],
    "new": [true, false, true]
}
```

Here's another example of nested containers:

```
[
    { "name": "X",
      "grams": 300,
      "qty": 4,
      "new": true },
    { "name": "Y",
      "grams": 200,
      "qty": 5,
      "new": false },
    { "name": "Z",
      "grams": 500,
      "qty": null,
      "new": true}
]
```

8.3.5 Data Table Toy Example

Let's consider a less basic example with some tabular data set:

Name	Gender	Homeland	Born	Jedi
Anakin	male	Tatooine	41.9BBY	yes
Amidala	female	Naboo	46BBY	no
Luke	male	Tatooine	19BBY	yes
Leia	female	Alderaan	19BBY	no
Obi-Wan	male	Stewjon	57BBY	yes
Han	male	Corellia	29BBY	no
Palpatine	male	Naboo	82BBY	no
R2-D2	unknown	Naboo	33BBY	no

How can we store this tabular data in JSON format? There are several ways to represent this data in JSON format. One option could be a JSON-array containing JSON-objects. Each JSON-object represents an individual:

```
    [
        {
         "Name": "Anakin",
         "Gender": "male",
```

```
        "Homeworld": "Tatooine",
        "Born": "41.9BBY",
        "Jedi": "yes"
       },
       {
        "Name": "Amidala",
        "Gender": "female",
        "Homeworld": "Naboo",
        "Born": 46BBY",
        "Jedi": "no"
       },
       ...
       {
        "Name": "R2-D2",
        "Gender": "unknown",
        "Homeworld": "Naboo",
        "Born": "33BBY",
        "Jedi": "no"
       }
   ]
```

Another way to represent the data in the table above is by using an object containing key-value pairs in which the *keys* are the names of the columns, and the *pairs* are arrays (the data values in each column).

```
{
  "Name": [ "Anakin", "Amidala", "Luke", ... , "R2-D2" ],
  "Gender": [ "male", "female", "male", ... , "unknown" ],
  "Homeworld": [ "Tatooine", "Naboo", "Tatooine", ... , "Naboo" ],
  "Born": [ "41.9BBY", "46BBY", "19BBY", ... , "33BBY" ],
  "Jedi": [ "yes", "no", "yes", ... , "no" ]
}
```

Chapter 9

JSON R packages

R has 3 packages for working with JSON data

- `"RJSONIO"` by Duncan Temple Lang
- `"rjson"` by Alex Couture-Beil
- `"jsonlite"` by Jeroen Ooms, Duncan Temple Lang, Jonathan Wallace

All packages provide 2 main functions, `toJSON()` and `fromJSON()`, that allow conversion **to** and **from** data in JSON format, respectively. We'll focus on the functions from `"jsonlite"`.

For illustration purposes, let us consider the package `"jsonlite"`.

There are 2 primary functions in `"jsonlite"`:

- `toJSON()` converts an R object to a string in JSON
- `fromJSON()` converts JSON content to R objects

9.1 Function `toJSON()`

The function `jsonlite::toJSON()` converts an R object to a string in JSON.

Example: single number to JSON-array

Let's begin with a super simple example by passing a single data value to the function `toJSON()`:

```
toJSON(pi, digits = 4)
#> [3.1416]
```

Example: vectors to JSON-arrays

Consider the following vectors

```
num <- c(1, 2, 3, 4, 5)
lts <- c('a', 'b', 'c', 'd', 'e')
```

Applying `toJSON()` to the vectors `num` and `lts` produces JSON arrays:

```
toJSON(num)
#> [1,2,3,4,5]

toJSON(lts)
#> ["a","b","c","d","e"]
```

The argument `pretty = TRUE` allows you to obtain a JSON string with added indentation whitespace:

```
toJSON(num, pretty = TRUE)
#> [1, 2, 3, 4, 5]

toJSON(lts, pretty = TRUE)
#> ["a", "b", "c", "d", "e"]
```

What about an R vector with named elements? For example, here's a vector `vec`

```
vec <- num
names(vec) <- lts
vec
#> a b c d e
#> 1 2 3 4 5
```

Converting `vec` to JSON, we get:

```
toJSON(vec)
#> [1,2,3,4,5]
```

As you can tell, the names of the elements in `vec` are lost in translation.

Example: matrix to JSON-array

Here's another example from an matrix to a JSON array:

```
mat <- matrix(9:1, nrow = 3, ncol = 3)
mat
#>      [,1] [,2] [,3]
#> [1,]    9    6    3
#> [2,]    8    5    2
#> [3,]    7    4    1
```

`toJSON()` converts an R matrix into a JSON-array

```
toJSON(mat)
#> [[9,6,3],[8,5,2],[7,4,1]]
```

Notice that the returned output arranges the values of the matrix row-by-row, also referred to as *row-major*. This means that when the input is an R matrix, `toJSON()` uses its argument `matrix = "rowmajor"`.

You can change the arrangement to *column-major* by specifying the argument `matrix = "columnmajor"`:

```
toJSON(mat, matrix = "columnmajor")
#> [[9,8,7],[6,5,4],[3,2,1]]
```

Example: data frame to JSON-object

We can also use `toJSON()` on data frames. Here's an example of an assembled data frame `swdf` which will be converted to a JSON-object:

```
# toy data
sw_data <- rbind(
  c("Anakin", "male", "Tatooine", "41.9BBY",  "yes"),
  c("Amidala", "female", "Naboo", "46BBY", "no"),
  c("Luke", "male", "Tatooine", "19BBY", "yes"),
  c("Leia", "female", "Alderaan", "19BBY", "no")
)

# convert to data.frame and add column names
swdf <- data.frame(sw_data, stringsAsFactors = FALSE)
names(swdf) <- c("Name", "Gender", "Homeworld", "Born", "Jedi")
swdf
#>      Name Gender Homeworld    Born Jedi
#> 1  Anakin   male  Tatooine 41.9BBY  yes
#> 2 Amidala female     Naboo   46BBY   no
#> 3    Luke   male  Tatooine   19BBY  yes
#> 4    Leia female  Alderaan   19BBY   no
```

The default output when you pass a data frame to `jsonlite::toJSON()` is

```
# convert R data.frame to JSON
sw_json = toJSON(swdf)
sw_json
#> [{"Name":"Anakin","Gender":"male","Homeworld":"Tatooine","Born":"41.9BBY","Jedi":"yes"},{"Name
```

The argument `dataframe` gives you more control on the output. This argument has three options:

- `"rows"`: each row is converted to a JSON-object with *key-value* pairs

formed by `"column_name": "row_value"`;

```
toJSON(swdf, dataframe = "rows")
#> [{"Name":"Anakin","Gender":"male","Homeworld":"Tatooine","Born":"41.9BBY","Jedi":"y
```

- `"columns"`: each column is converted into a JSON-object with a single *key* for each column, and *values* stored as arrays;

```
toJSON(swdf, dataframe = "columns")
#> {"Name":["Anakin","Amidala","Luke","Leia"],"Gender":["male","female","male","female
```

- `"values"`: the values in each column are converted to a JSON-array, and the names of the columns are lost.

```
toJSON(swdf, dataframe = "values")
#> [["Anakin","male","Tatooine","41.9BBY","yes"],["Amidala","female","Naboo","46BBY","
```

9.2 Function `fromJSON()`

In practice, instead of converting R objects to JSON objects, it is more common to have data in JSON format which needs to be converted into an R object.

The function `jsonlite::fromJSON()` converts a JSON-object to an R object.

Example: JSON-array to R vector

```
json_array <- '["computing", "with", "data"]'

fromJSON(json_array)
#> [1] "computing" "with"      "data"
```

Example: JSON-object to R object

Consider a simple JSON-object, and its conversion to R with `jsonlite::fromJSON()`

```
json_obj1 <- '{"name": "Jessica"}'

fromJSON(json_obj1)
#> $name
#> [1] "Jessica"
```

Notice that the obtained object is an R list in which the *key* becomes the name of the list, and the *value* becomes the content of the list's element.

Consider a less simple JSON-object:

```
json_obj2 <- '{"name1": "Nicole", "name2": "Pleuni", "name3": "Rori"}'

fromJSON(json_obj2)
```

```
#> $name1
#> [1] "Nicole"
#>
#> $name2
#> [1] "Pleuni"
#>
#> $name3
#> [1] "Rori"
```

Another example:

```
fromJSON('{"name": ["X", "Y"], "grams": [30, 20], "qty": [4, null],
"new": [true, false]}')
#> $name
#> [1] "X" "Y"
#>
#> $grams
#> [1] 30 20
#>
#> $qty
#> [1]  4 NA
#>
#> $new
#> [1]  TRUE FALSE
```

Example: JSON-object to R object

Suppose you have a JSON object with the following data:

```
{
  "Name": ["Anakin","Amidala","Luke","Leia"],
  "Gender": ["male","female","male","female"],
  "Homeworld": ["Tatooine","Naboo","Tatooine","Alderaan"],
  "Born": ["41.9BBY","46BBY","19BBY","19BBY"],
  "Jedi": ["yes","no","yes","no"]
}
```

and assume that the above data is stored as a single (continuous) string in an R character vector `json_sw`; applying `fromJSON()` to this string gives you the following list:

```
fromJSON(json_sw)
#> $Name
#> [1] "Anakin"  "Amidala" "Luke"    "Leia"
#>
#> $Gender
#> [1] "male"   "female" "male"   "female"
```

```
#>
#> $Homeworld
#> [1] "Tatooine" "Naboo"    "Tatooine" "Alderaan"
#>
#> $Born
#> [1] "41.9BBY" "46BBY"   "19BBY"   "19BBY"
#>
#> $Jedi
#> [1] "yes" "no"  "yes" "no"
```

Can this be transformed into a data frame? Yes, by passing the obtained list to the function `data.frame()`:

```
data.frame(fromJSON(json_sw))
#>      Name Gender Homeworld    Born Jedi
#> 1  Anakin   male  Tatooine 41.9BBY  yes
#> 2 Amidala female     Naboo   46BBY   no
#> 3    Luke   male  Tatooine   19BBY  yes
#> 4    Leia female  Alderaan   19BBY   no
```

9.3 Reading JSON Data

Now that we have discussed the basics of JSON, and the common ways to convert `fromJSON()` and `toJSON()`, let's see how to read JSON data from the Web.

One of the typical ways to import JSON data from the Web to R is by passing the url directly to `fromJSON()`. Another way is by passing the name of the file with the JSON content as a single string to the function `fromJSON()`.

Here's an example reading a JSON string from the website *Advice Slip*. The url https://api.adviceslip.com/advice gives you a random advice (see figure below):

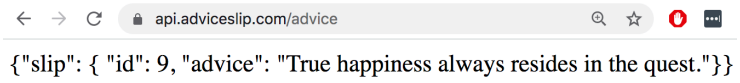


Figure 9.1: Random advice from Advice Slip

As you can tell, the content is a simple JSON string

```
advice_url <- "https://api.adviceslip.com/advice"

fromJSON(advice_url)
```

```
#> $slip
#> $slip$id
```

```
#> [1] 9
#>
#> $slip$advice
#> [1] "True happiness always resides in the quest!"
```

Example: Colors in Hexadecimal Notation

The following data comes from one of Dave Eddy's github repositories:

https://raw.githubusercontent.com/bahamas10/css-color-names/master/css-color-names.json

This is a JSON-object in which the *keys* are color-names, and the *values* are the hexadecimal digits of the corresponding color:

```
{
  "aliceblue": "#f0f8ff",
  "antiquewhite": "#faebd7",
  "aqua": "#00ffff",
  "aquamarine": "#7fffd4",
  "azure": "#f0ffff",
  ...
  "wheat": "#f5deb3",
  "white": "#ffffff",
  "whitesmoke": "#f5f5f5",
  "yellow": "#ffff00",
  "yellowgreen": "#9acd32"
}
```

We pass the url to `jsonlite::fromJSON()`

```
colors_json <- "https://raw.githubusercontent.com/bahamas10/css-color-names/master/css-color-name

hex_colors <- fromJSON(colors_json)
```

The output in `hex_colors` is a list with 148 elements; the first five elements are displayed below:

```
hex_colors[1:5]
#> $aliceblue
#> [1] "#f0f8ff"
#>
#> $antiquewhite
#> [1] "#faebd7"
#>
#> $aqua
#> [1] "#00ffff"
#>
#> $aquamarine
```

```
#> [1] "#7fffd4"
#>
#> $azure
#> [1] "#f0ffff"
```

Part V

APIs

Chapter 10

Web APIs

In this chapter we'll give you a crash introduction to Web APIs, and how to use R for interacting with them.

You will need the following packages

```
library(httr2)
library(xml2)
library(jsonlite)
```

10.1 Introduction

So far we've been dealing with data sets in various formats: internal data objects in R (e.g. data tibble `starwars`), built-in data frames such as `mtcars` or `oldfaithful`), reading files stored in your computer (txt, csv, tsv, etc). But you also need to learn how to get data from the web.

For better or worse, reading data from the Web entails a whole other set of considerations. Because of the large variety of data formats available in the Web, we will primarily focus on retrieving data from *Application Programming Interfaces* also known as APIs.

The reason to focus on APIs is because nowadays many companies, websites, sources, etc. use APIs as their primary means to share information and data. Many large websites like Reddit, Twitter and Facebook offer APIs so that data analysts and data scientists can access interesting data. And having an API to share data has become a standard thing to have.

10.2 A little bit about APIs

API stands for **Application Programming Interface**. If this sounds too fancy or cryptic for you, then simply think of it as a "Data Sharing Interface".

Instead of having to download a data file, an API allows programmers to request data directly from a website. From a technical point of view, an API is a set of rules, protocols, and tools for building software and applications.

What is an API?

"API" is a general term for the place where one computer program (the client) interacts with another (the server), or with itself.

APIs offer data scientists a polished way to request clean and curated data from a website. When a website like Facebook sets up an API, they are essentially setting up a computer that waits for data requests.

Once this computer receives a data request, it will do its own processing of the data and send it to the computer that requested it. From our perspective as the requester, we will need to write code in R that creates the request and tells the computer running the API what we need. That computer will then read our code, process the request, and return nicely-formatted data that can be easily parsed by existing R libraries.

Why to use an API?

Why is this valuable? Contrast the API approach to pure web scraping. When a programmer scrapes a web page, they receive the data in a messy chunk of HTML. While there are certainly libraries out there that make parsing HTML text easy, these are all cleaning steps that need to be taken before we even get our hands on the data we want!

Often, we can immediately use the data we get from an API, which saves us time and frustration.

10.3 Using R as an HTTP Client

R has a few HTTP client packages: `"crul"`, `"curl"`, `"httr2"`, and `"RCurl"`; you can think of them as "high-level R HTTP clients" which basically let you use R (and your computer) as an **HTTP client**.

We will describe how to use functions from `"httr2"` (pronounced *hitter2*).

10.4 Interacting with AP's via R

In R, we can use the `"httr2"` package to make http requests and handle the responses.

Let's start with baby steps using the website https://api.adviceslip.com/ which provides an API to get a free piece of advice from the internet.

The first thing you need to do is to look at the web page to familiarize yourself with the functionalities it provides.

Figure 10.1: Advice Slip JSON API

The url `https://api.adviceslip.com/advice` will give you a random advice:

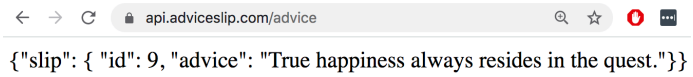


Figure 10.2: Random advice from Advice Slip

10.4.1 Making request from R

Notice that the format of the response is provided in JSON. In the above example, the advice is given as:

```
{"slip": {"id": 9, "advice": "True happiness always resides in the quest."}}
```

which we can mentally rearrange as:

```
{
  "slip": {
    "id": 9,
    "advice": "True happiness always resides in the quest."
  }
}
```

Getting a random advice is quite simple, all you need is to make an HTTP request using the url `https://api.adviceslip.com/advice`.

Interestingly, we can make such request from R, using it as a server. This requires employing some functions from `"httr2"`.

In `"httr2"`, you start by creating a **request**. How? You use the `request()` function which creates a request object. To be clear, `request()` does not submit the request, it only creates the object, which you can use to build up a complex request piece by piece, and works well with the pipe operators `|>` or `%>%`.

```
# start a request object
advice_url = "https://api.adviceslip.com"
req = request(advice_url)
req
#> <httr2_request>
#> GET https://api.adviceslip.com
#> Body: empty
```

To see what this request will send to the server we perform a dry run:

```
req |> req_dry_run()
#> GET / HTTP/1.1
#> Host: api.adviceslip.com
#> User-Agent: httr2/1.0.0 r-curl/5.1.0 libcurl/7.79.1
#> Accept: */*
#> Accept-Encoding: deflate, gzip
```

What's going on with the first line `GET / HTTP/1.1`?

- The first term, `GET`, refers to the HTTP **method**, which is a verb that tells the server what you want to do. In this case is `GET`, the most common verb, indicating that we want to get a resource. Other verbs include `POST`, to create a new resource; `PUT`, to replace an existing resource; and `DELETE`, to delete a resource.
- The second part, `/`, is the **path** which is the URL stripped of details that the server already knows, i.e. the protocol (http or https), and the host (localhost).
- The third element, `HTTP/1.1`, is the version of the HTTP protocol. This is unimportant for our purposes because it's handled at a lower level.

In order to make a request with `"httr2"`, we need to complete the full path of the URL. In the above example, this is done with `req_url_path_append()`

```
# then we complete the full path
req |>
  req_url_path_append("advice")
#> <httr2_request>
#> GET https://api.adviceslip.com/advice
#> Body: empty
```

10.4.2 Performing a Request

Once we have the desired request object, then we can submit it or **perform** such request with `req_perform()`:

```
# then add on the query path
resp = req |>
  req_url_path_append("advice") |>
  req_perform()

resp
#> <httr2_response>
#> GET https://api.adviceslip.com/advice
#> Status: 200 OK
#> Content-Type: text/html
#> Body: In memory (77 bytes)
```

As you can tell, the response has a success status (`200 OK`), and the fetched content is text in html format.

The object `resp` is an object of class `"httr2_response"`, which is basically an R list that contains 7 elements:

```
names(resp)
#> [1] "method"      "url"         "status_code" "headers"     "body"
#> [6] "request"     "cache"
```

As you can tell, one of the elements in `resp` is `"body"`. If we take a look at the body we get an interesting—but unhelpful—output:

```
resp$body
#>  [1] 7b 22 73 6c 69 70 22 3a 20 7b 20 22 69 64 22 3a 20 39 2c 20 22 61 64 76
#> [25] 69 63 65 22 3a 20 22 54 72 75 65 20 68 61 70 70 69 6e 65 73 73 20 61 6c
#> [49] 77 61 79 73 20 72 65 73 69 64 65 73 20 69 6e 20 74 68 65 20 71 75 65 73
#> [73] 74 2e 22 7d 7d
```

What kind of object is `resp$body`? Inspecting its `class()`, it turns out that this is an object of class `"raw"` or *raw vector*, which is a vector that holds raw bytes in R.

```
class(resp$body)
#> [1] "raw"
```

Technically speaking, a raw vector is printed with each byte separately represented as a pair of hex digits. If you want to see a character representation (with escape sequences for non-printing characters) use `rawToChar()`.

```
body_json = rawToChar(resp$body)

body_json
```

```
#> [1] "{\"slip\": { \"id\": 9, \"advice\": \"True happiness always resides in the que
```

Converting the raw vector into text, we obtain the response body `body_json` which is text in JSON format. Then, to parse this JSON text, we use `fromJSON()` which returns an R list:

```
slip_advice = fromJSON(body_json)

slip_advice
#> $slip
#> $slip$id
#> [1] 9

#> $slip$advice
#> [1] "True happiness always resides in the quest."
```

And finally, we extract the piece of advice as follows:

```
slip_advice$slip$advice
#> [1] "True happiness always resides in the quest."
```

10.4.3 Extracting response as string

Let me show you another more straightforward way to extract the content of the response body by using one the `resp_body_()` functions, in particular the `resp_body_string()` function:

```
resp |> resp_body_string()
#> [1] "{\"slip\": { \"id\": 9, \"advice\": \"True happiness always resides in the que
```

Notice that we obtain the same JSON text, which we can parse with `fromJSON()`

```
resp |> resp_body_string() |> fromJSON()
#> $slip
#> $slip$id
#> [1] 9

#> $slip$advice
#> [1] "True happiness always resides in the quest."
```

10.4.4 Extracting response as HTML

A third equivalent method to extract the response body is with the function `resp_body_html()`. The reason shy we can do that in this case has to do with the fact that the content of the response is in HTML format: `Content-Type: text/html`.

```
# the result comes back as html
# Let's extract body from response
doc_html = resp |> resp_body_html()

doc_html
#> {html_document}
#> <html>
#> [1] <body><p>{"slip": { "id": 9, "advice": "True happiness always resides ...
```

Since the output is an `html_document`, we first need to extract the text of the `<p>` element. One option to achieve this is some xpath expression:

```
txt_json = doc_html |>
  xml_find_all(xpath = "//p") |>
  xml_text()

txt_json
#> [1] "{\"slip\": { \"id\": 9, \"advice\": \"True happiness always resides in the quest.\"}}"
```

This still requires some JSON manipulation:

```
slip_advice = fromJSON(txt_json)
slip_advice
#> $slip
#> $slip$id
#> [1] 9

#> $slip$advice
#> [1] "True happiness always resides in the quest."
```

10.5 Example: Search with Advice ID

The Advice Slip API also allows you to request an advice based on its ID.

Advice by ID

HTTP Method	GET
URL	https://api.adviceslip.com/advice/{slip_id}
Description	If an advice slip is found with the corresponding {slip_id}, a slip object is returned.
Parameters	callback string To define your own callback function name and return the JSON in a function wrapper (as JSONP), add the parameter callback with your desired name as the value.

Figure 10.3: Random advices by id

For example, the id = 5 results in the following advice:

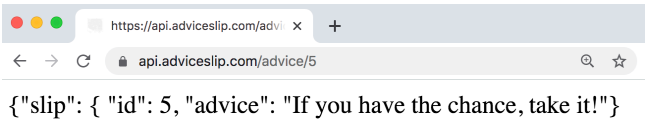


Figure 10.4: Random advice from Advice Slip

To make this request from R, we create the request object, and then perform such request

```
# advice id=5
resp_advice_id5 <- req |>
  req_url_path_append("advice/5") |>
  req_perform()

resp_advice_id5
#> <httr2_response>
#> GET https://api.adviceslip.com/advice/5
#> Status: 200 OK
#> Content-Type: text/html
#> Body: In memory (66 bytes)
```

We can then extract the response body, for instance:

```
advice_id5 = resp_advice_id5 |> resp_body_string() |> fromJSON()
advice_id5
#> $slip
#> $slip$id
#> [1] 5

#> $slip$advice
#> [1] "If you have the chance, take it!"
```

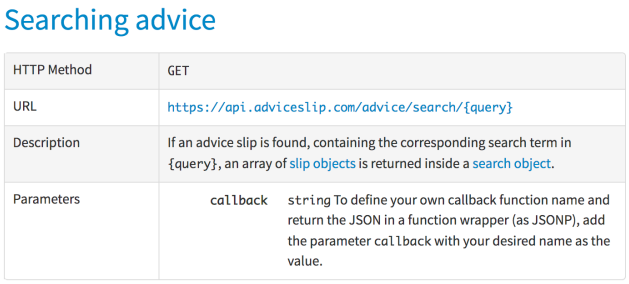
10.6 Example: Search Query

Another kind of request that you can do with the Advice Slip API is to search for an advice specifying a search query:

For example, say we are interested in searching for advice that includes the word **chance**. The corresponding URL path will have the following form:

`https://api.adviceslip.com/advice/search/chance`

Let's search for the term `chance`. For sake of illustration, let's build a request object by appending the path elements: `advice`, `search`, and `chance`



Searching advice

HTTP Method	GET
URL	https://api.adviceslip.com/advice/search/{query}
Description	If an advice slip is found, containing the corresponding search term in {query}, an array of slip objects is returned inside a search object.
Parameters	callback string To define your own callback function name and return the JSON in a function wrapper (as JSONP), add the parameter callback with your desired name as the value.

Figure 10.5: Random advices with search query

{"total_results": "3", "query": "chance", "slips": [{"id":5,"advice":"If you have the chance, take it!","date":"2016-12-25"},{"id":184,"advice":"You can fail at what you don't want. So you might as well take a chance on doing what you love.","date":"2017-03-10"},{"id":185,"advice":"You can fail at what you don't want. So you might as well take a chance on doing what you love.","date":"2016-08-05"}]}

Figure 10.6: Random advices with search query 'chance'

```
# advice id=5
resp_advice_chance <- req |>
  req_url_path_append("advice") |>
  req_url_path_append("search") |>
  req_url_path_append("chance") |>
  req_perform()
```

The above command can be shorten as:

```
# advice id=5
resp_advice_chance <- req |>
  req_url_path_append("advice/search/chance") |>
  req_perform()

resp_advice_chance
#> <httr2_response>
#> GET https://api.adviceslip.com/advice/search/chance
#> Status: 200 OK
#> Content-Type: text/html
#> Body: In memory (402 bytes)
```

Having performed the request, we fetch the data by extracting the body as a string (in JSON format), and then parsing with `fromJSON()`

```
advice_chance = resp_advice_chance |> resp_body_string() |> fromJSON()
```

```
names(advice_chance)
#> [1] "total_results" "query"         "slips"
```

The "slips" element contains a data frame with 3 advice recommendations:

```
advice_chance$slips
#>    id
#> 1   5
#> 2 184
#> 3 185
#>
#> 1                                                                    If you have the ch
#> 2 You can fail at what you don't want. So you might as well take a chance on doing t
#> 3 You can fail at what you don't want. So you might as well take a chance on doing t
#>         date
#> 1 2016-12-25
#> 2 2017-03-10
#> 3 2016-08-05
```

Chapter 11

PubMed API Example

In this chapter, we provide an example of web data collection from the database PubMed, using the **Entrez Programming Utilities**, commonly referred to as E-utilities, from the National Center for Biotechnology Information (NCBI).

You will need the following packages:

```
library(stringr)    # for strings and regular expressions
library(xml2)       # for parsing data in XML (e.g. HTML)
library(rvest)      # for scraping XML and HTML content
library(tm)         # for text mining
library(wordcloud2) # for graphing wordclouds (optional)
```

11.1 PubMed

PubMed (https://www.pubmed.gov) is a database of the largest collection of citations to medical journal literature in the world, and it is one of 38 databases built and maintained by the NCBI. Scientists, researchers, and users around the world use PubMed to search and retrieve bibliographic data, choose from several display formats, and share their results. Keep in mind that when people talk about PubMed, they could be referring to both the search interface and to the database itself.

PubMed's website provides a search engine to obtain bibliographic information:

The simplest use of PubMed's search engine is to provide a query, very similar to the queries that you would provide to google's search engine. For example, we may be interested in looking for articles and other publications associated with some of the effects that exposure to Bisphenol A (BPA) has on reproductive health. Therefore, we can type in `BPA exposure reproductive health` inside the search box, and obtain some results (like those displayed in the screenshot

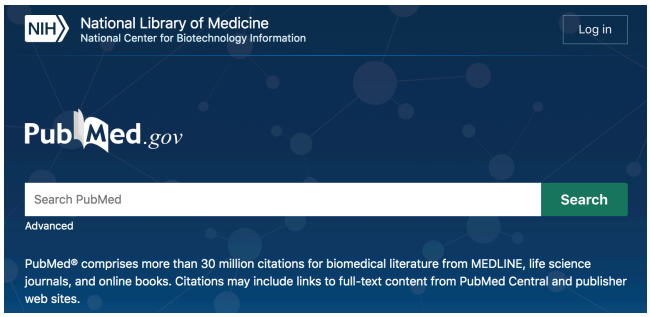


Figure 11.1: Partial screenshot of PubMed's homepage

below).

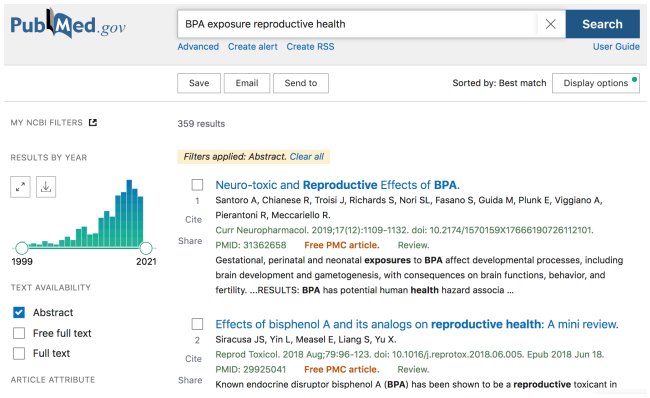


Figure 11.2: PubMed search example

As of this writing (Fall 2020), there are 359 results that match the query term, with publications ranging from 1999 to end-of-2020. Notice that the webpage has a sidebar with checkboxes, and other intereactive options, that allow you to filter results by year of publication, by searching for the query in the abstract, or just in the titles, and things like that. As an example, we can move the slider for the year of publication to retrieve results that were published in 2018 and 2019 (see screenshot below)

If the obtained results are what you were looking for, you also have the option to download a CSV file with such results (download button in the navigation bar, above the barchart of years of publication).

In addition, you can perform a more advanced search by clicking on the "Advanced" button displayed below the search box. Clicking on this option will take you to a new page with more query boxes and a long list of query fields (screenshot below).

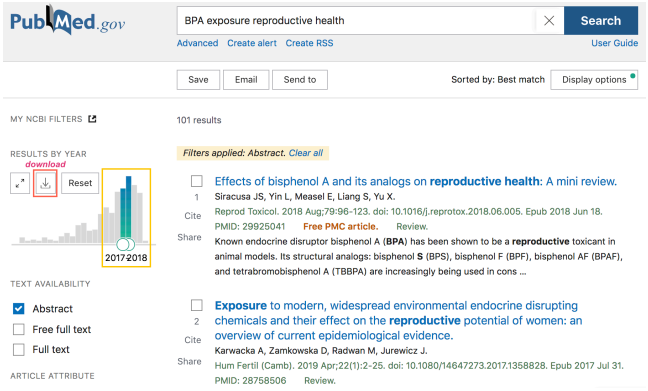


Figure 11.3: PubMed search example (cont'd)

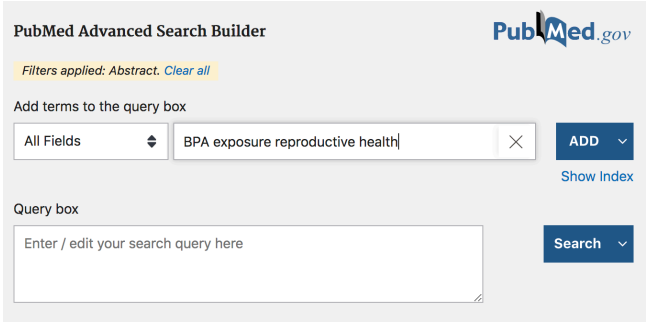


Figure 11.4: PubMed search example (cont'd)

In the Advanced search mode, you can find a large list of fields that give you the opportunity to specify a more detailed query. For example, you can be more specific in your search by looking for results based on title, or based on author(s) information, or by date of publication, etc.

This way of interacting with PubMed (and similar databases) is how most users and researchers utilize PubMed: performing manual searches through their browsers, obtaining the results on their screens, and deciding which publications are worth further inspection. Sometimes, however, you may:

- have a question that cannot be answered easily when you search PubMed,
- need to see every citation in PubMed in a certain field,
- need to run a search in PubMed and get the output in a CSV file that includes more (or different) data elements than the standard CSV file,
- need to run specialized queries that might serve a very specific research need.

This is where **E-utilities** comes very handy because you can obtain the data that you need, and only the data that you need, in the format that you need. E-utilities is a great solution when you:

- can't find a good way to ask your question using the PubMed search box.
- can search PubMed, but you'd like more, or less, or different data returned from the records.
- want more control over the format of your PubMed data.

E-utilities is simply another way to search PubMed and the other NCBI databases. Formally, E-utilities is an Application Programming Interface (API) which allows you to control exactly what fields you are searching, the specific data elements you retrieve, the format of the data, and how you share your results. When you use E-utilities to access PubMed, you are accessing the same data that you would find at https://www.pubmed.gov.

11.2 Basics of E-utilities

E-utilities is the short name for the **Entrez Programming Utilities**, which is simply another way to search PubMed and other NCBI databases. The E-utilities website is:

https://www.ncbi.nlm.nih.gov/books/NBK25500/

which contains the official documentation, written and maintained by Eric Sayers.

According to the website:

> "Entrez Programming Utilities (E-utilities) are a set of nine server-side programs that provide a stable interface into the Entrez query and database system at the National Center for Biotechnology Information (NCBI)."

What does this mean? E-utilities is basically an Application Programming Interface (API). The E-utilities API allows you to search PubMed and any other NCBI database through your own program: e.g. R, Python, etc. When you use E-utilities to access PubMed, you are accessing the same data that you'd find at https://www.pubmed.gov.

Note: the results returned by E-utilities queries of PubMed may differ slightly from those returned in the web version of PubMed. As of this writing (Fall 2020) a new PubMed API is currently under development.

The Nine E-utilities

The name *E-utilities* comes from the nine utilities (or programs):

- **EInfo** (database statistics): Provides the number of records indexed in each field of a given database, the date of the last update of the database, and the available links from the database to other Entrez databases.
- **ESearch** (database statistics): Responds to a text query with the list of matching UIDs in a given database (for later use in ESummary, EFetch or ELink), along with the term translations of the query.
- **EPost** (UID uploads): Accepts a list of UIDs from a given database, stores the set on the History Server, and responds with a query key and web environment for the uploaded dataset.
- **ESummary** (document summary downloads): Responds to a list of UIDs from a given database with the corresponding document summaries.
- **EFetch** (data record downloads): Responds to a list of UIDs in a given database with the corresponding data records in a specified format.
- **ELink** (Entrez links): Responds to a list of UIDs in a given database with either a list of related UIDs (and relevancy scores) in the same database or a list of linked UIDs in another Entrez database
- **EGQuery** (global query): Responds to a text query with the number of records matching the query in each Entrez database.
- **ESpell** (spelling suggestions): Retrieves spelling suggestions for a text query in a given database.
- **ECitMatch** (batch citation searching in PubMed): Retrieves PubMed IDs (PMIDs) corresponding to a set of input citation strings.

For illustration purposes, we will only focus on ESearch, ESummary, and EFetch.

11.2.1 How does E-utilities work?

The way you use E-utilities is by assembling an e-utilities URL, following a specific set of rules, that you can use to make a request to one of its nine servers.

Behind the scenes, the assembled E-utilities URL is translated into a standard set of input parameters that are used as the values necessary for various NCBI software components to search for and retrieve the requested data. In other words, the URLs direct requests to servers that are used only by the E-utilities and that are optimized to give users the best performance.

Before making any requests, keep in mind the following recommendation:

> "In order not to overload the E-utility servers, NCBI recommends that users post no more than three URL requests per second and limit large jobs to either weekends or between 9:00 PM and 5:00 AM Eastern time during weekdays."

By the way, you can obtain an API Key offering you enhanced levels of supported access to the E-utilities. This is totally optional but worth knowing. In our examples, we won't need an API key, but if you plan to use PubMed more intensively then consider getting an API Key.

API Requests

To make requests, you have to assemble a URL. Each URL consists of three parts:

1) **The base URL**: This is the address of the E-utilities server. Every URL begins with `https://eutils.ncbi.nlm.nih.gov/entrez/eutils/`

2) **A utility name**: This is the name of the specific tool that you are using. There are nine E-utilities and each one performs a specific function:

   - **ESearch**: Search a text query in a single database and retrieve the list of matching unique identifiers (UIDs). In PubMed, ESearch retrieves a list of PMIDs. `esearch.fcgi?`
   - **ESummary**: Retrieve document summaries for each UID. `esummary.fcgi?`
   - **EFetch**: Retrieve full records for each UID. `efetch.fcgi?`
   - **EPost**: Upload a list of UIDs for later use. `epost.fcgi?`
   - **ELink**: Retrieve UIDs for related or linked records, or LinkOut URLs. `elink.fcgi?`
   - **EInfo**: Retrieve information and statistics about a single database. `einfo.fcgi?`
   - **ESpell**: Retrieve spelling suggestions for a text query. `espell.fcgi?`
   - **ECitMatch**: Search PubMed for a series of citation strings. `ecitmatch.fcgi?`

- **EGQuery**: Search a text query in all databases and return the number of results for the query in each database. `egquery.fcgi?`

The file extension `.fcgi?` stands for *Fast Common Gateway Interface*.

3) **The parameters**: These are the details of your query. Common parameters include the name of the database, your search terms, the number of results you would like to get, and the format of the output. The parameters that are available will change depending on the utility.

When you put these three parts together, you will have a URL that looks something like this:

`https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=pubmed&term=reproductive+AND+health`

Let's review three examples using utilities ESearch, ESummary, and EFetch.

Example: Searching PubMed with ESearch

Suppose we are interested in searching for bibliographic information in PubMed for the term *reproductive health*. Typically, the first thing to do when searching for information in PubMed is to retrieve a list of unique IDs for the documents that match the query text. All of this is done with **ESearch**, which means we need to use the `esearch.fcgi?` path to assemble the URL:

`https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=pubmed&term=reproductive+AND+health`

The above url uses the ESearch utility (`esearch.fcgi?`) and uses two query parameters, namely `db` and `term`.

- `https://eutils.ncbi.nlm.nih.gov/entrez/eutils/` is the base URL
- `esearch.fcgi?` indicates the E-utility (*ESearch* in this case)
- `db=pubmed`: indicates that the searched database is PubMed.
- `term=reproductive+AND+health`: indicates that the searched involves the term *reproductive health*.
- notice the use of `&` to include a new query parameter

By default, the information is retrieved in XML format (see screenshot below). If you scroll through the XML output you'll see the number of records retrieved along with a list of unique document PubMed IDs (PMIDs) for those records.

Example: Retrieving records with EFetch

The **ESearch** URL retrieves a list of PMIDs—not full records. To get the full records you need to use **EFetch**—providing the list of IDs—and the URL would look something like this:

`https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&id=33073741,33073726&retmode=`

- `https://eutils.ncbi.nlm.nih.gov/entrez/eutils/` is the base URL

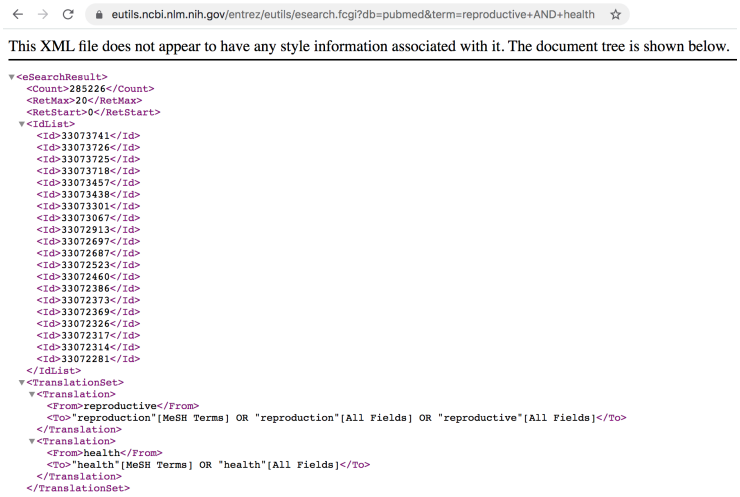


Figure 11.5: List of PubMed IDs in XML format

- `efetch.fcgi?` indicates the E-utility (*EFectch* in this case)
- `db=pubmed`: indicates that the searched database is PubMed.
- `id=33073741,33073726`: includes the PMIDs of the records.
- `retmode=abstract`: indicates the `abstract` as the *return mode*.
- `rettype=text`: determines the `text` as the *return type* format.

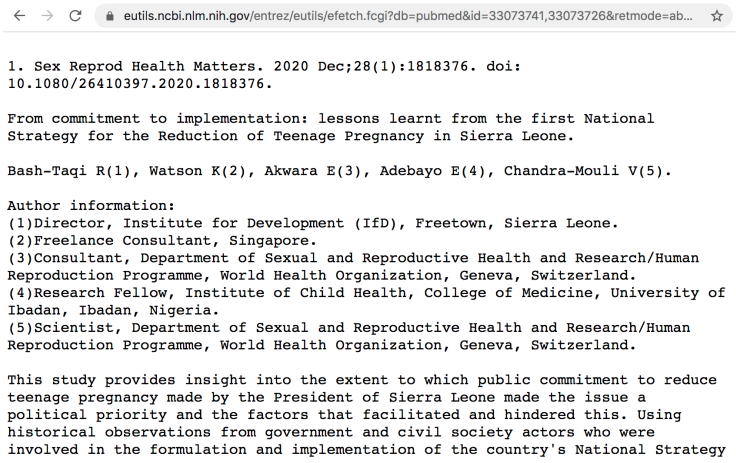


Figure 11.6: Return type in text format for abstract of specified IDs

Example: Retrieving summaries with ESummary

The **ESummary** URL allows you to get the document summaries of the associated list of PMIDs, and the URL would look something like this:

`https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi?db=pubmed&id=33073741,33073726`

- `https://eutils.ncbi.nlm.nih.gov/entrez/eutils/` is the base URL
- `esummary.fcgi?` indicates the E-utility (*ESummary* in this case)
- `db=pubmed`: indicates that the searched database is PubMed.
- `id=33073741,33073726`: includes the PMIDs of the records.

The output of ESummary is a series of XML `"DocSums"` (Document Summaries), the format of which depends on the database. Below is an example DocSum for PubMed ID 33073741.

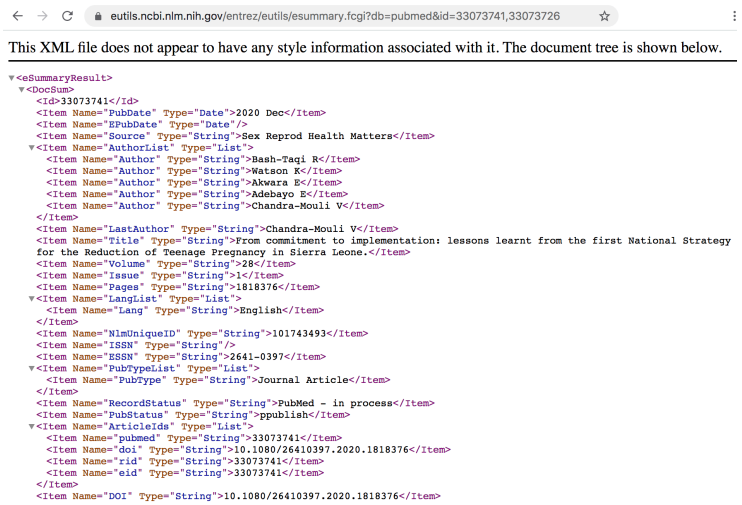


Figure 11.7: Summary in XML format for specified IDs

11.2.2 Searching PubMed from within R

Now that you have a basic understanding of PubMed, and the E-Utilities programs, let's see how to make requests from R.

We are going to consider a typical pipeline that involves three steps:

1) use ESearch to get a list of document IDs (default output in XML)
2) parse the IDs from the XML document
3) pass the list of IDs to either EFetch or ESummary

Step 1: As a first step, we can define a character string `entrez_url` with the base URL, and three more strings for each of the E-utilities: `esearch`, `efetch`, and `esummary`:

```
# base URL, and paths of associated e-utilities
entrez_url <- "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/"

esearch <- "esearch.fcgi?"
efetch <- "efetch.fcgi?"
esummary <- "esummary.fcgi?"
```

Suppose we are interested in performing a search with the following parameters:

- searching for the term *BPA exposure reproductive health*;
- limiting our search to documents between 2018 and 2019
- retaining at most the first 100 results

Step 2: We can assemble an R string `params_esearch` with all the previous ESearch query parameters:

```
# assembling string of parameters for the query
params_esearch <- paste0(
  c("db=pubmed",
    "term=BPA+exposure+reproductive+health",
    "mindate=2018",
    "maxdate=2019",
    "retmax=100"),
  collapse = "&")

params_esearch
#> [1] "db=pubmed&term=BPA+exposure+reproductive+health&mindate=2018&maxdate=2019&retm
```

Step 3: With the base URL, the `esearch` utility, and the parameters, we assemble the required URL that we can pass to `read_xml()`, and then use `html_nodes()` and `html_text()` to extract the list of documents IDs:

```
# assemble an esearch URL
ids_query <- paste0(entrez_url, esearch, params_esearch)

# Retrieving IDs with ESearch
ids_xml <- read_xml(ids_query)

# extract Ids (PMIDs)
ids <- html_text(html_nodes(ids_xml, xpath = "//Id"))
```

The vector `ids` contains 94 retrieved document IDs, the first 5 and the last 5 five displayed below:

```
head(ids, 5)
#> [1] "31771501" "31697385" "31683046" "31658598" "31648075"

tail(ids, 5)
#> [1] "29549734" "29428396" "29415642" "29385186" "29317319"
```

Step 4: We can then use this list of IDs to assemble a URL for obtaining summary information of documents—via ESummary. Notice that Esummary uses different parameters from the ones used in ESearch. The query parameters in this case are `db` (name of database) and `id` (list of IDs). With the base URL, the `esummary` utility, and the parameters `params_esummary`, we assemble the required URL that we can pass to `read_xml()`:

```
# parameters for esummary
params_esummary <- paste0(
  "db=pubmed&id=",
  paste(ids, collapse = ","))

# assemble an esummary URL
summaries_query <- paste0(entrez_url, esummary, params_esummary)

summaries_xml <- read_xml(summaries_query)
```

Step 5: To inspect the XML output we can use functions from package `"xml2"` such as `xml_child()` for instance. Let's take a look at the first node:

```
xml_child(summaries_xml, search = 1)
#> {xml_node}
#> <DocSum>
#>  [1] <Id>31771501</Id>
#>  [2] <Item Name="PubDate" Type="Date">2019 Oct</Item>
#>  [3] <Item Name="EPubDate" Type="Date"/>
#>  [4] <Item Name="Source" Type="String">Toxicol Ind Health</Item>
#>  [5] <Item Name="AuthorList" Type="List">\n  <Item Name="Author" Type="String ...
#>  [6] <Item Name="LastAuthor" Type="String">Sun Z</Item>
#>  [7] <Item Name="Title" Type="String">Oral exposure to low-dose bisphenol A i ...
#>  [8] <Item Name="Volume" Type="String">35</Item>
#>  [9] <Item Name="Issue" Type="String">10</Item>
#> [10] <Item Name="Pages" Type="String">647-659</Item>
#> [11] <Item Name="LangList" Type="List">\n  <Item Name="Lang" Type="String">En ...
#> [12] <Item Name="NlmUniqueID" Type="String">8602702</Item>
#> [13] <Item Name="ISSN" Type="String">0748-2337</Item>
#> [14] <Item Name="ESSN" Type="String">1477-0393</Item>
#> [15] <Item Name="PubTypeList" Type="List">\n  <Item Name="PubType" Type="Stri ...
#> [16] <Item Name="RecordStatus" Type="String">PubMed - indexed for MEDLINE</Item>
#> [17] <Item Name="PubStatus" Type="String">ppublish</Item>
```

```
#> [18] <Item Name="ArticleIds" Type="List">\n  <Item Name="pubmed" Type="String ...
#> [19] <Item Name="DOI" Type="String">10.1177/0748233719885565</Item>
#> [20] <Item Name="History" Type="List">\n  <Item Name="entrez" Type="Date">201 ...
#> ...
```

The obtained summary information contains various fields such as date of publication `PubDate`, list of authors `AuthorList`, title `Title`, starting and ending pages `Pages`, etc.

Step 6: To extract all the titles, we look for the `Item` nodes with argument `Name="Title"`. To be more precise, we can define an XPath pattern `'//Item[@Name="Title"]'` for `xml_nodes()`, and then extract the content with `xml_text()`:

```
title_nodes <- xml_nodes(summaries_xml, xpath = '//Item[@Name="Title"]')
#> Warning: `xml_nodes()` was deprecated in rvest 1.0.0.
#> i Please use `html_elements()` instead.
#> This warning is displayed once every 8 hours.
#> Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
#> generated.
titles <- xml_text(title_nodes)
head(titles)
#> [1] "Oral exposure to low-dose bisphenol A induces hyperplasia of dorsolateral pros
#> [2] "Juvenile Toxicity Rodent Model to Study Toxicological Effects of Bisphenol A (
#> [3] "Bisphenol F exposure impairs neurodevelopment in zebrafish larvae (Danio rerio
#> [4] "Effects of Dietary Bisphenol A on the Reproductive Function of Gilthead Sea Br
#> [5] "Urinary bisphenol A concentration is correlated with poorer oocyte retrieval a
#> [6] "In utero exposure to persistent and nonpersistent endocrine-disrupting chemica
```

Finally, we can make a wordcloud. The code below has a series of commands to perform some text processing (with functions from the text mining package `"tm"`) with the main terms or words used to create the wordcloud.

```
# title words
twords <- unlist(str_split(titles, " "))
twords <- tolower(twords)

# auxiliary vector of stop words
stop_words <- c(
  stopwords(kind = "en"),
  c("bisphenol", "exposure", "reproductive", "health", "bpa")
)

# removing unnecessary things
twords <- removePunctuation(twords)
twords <- removeNumbers(twords)
twords <- removeWords(twords, stopwords(kind = "en"))
```

```
twords <- removeWords(twords, stop_words)
twords <- twords[twords != ""]

# data frame of terms and frequencies
dat_words <- as.data.frame(table(twords))
dat_words <- dat_words[order(dat_words$Freq, decreasing = TRUE), ]
head(dat_words, n = 10)
#>             twords Freq
#> 161        effects   18
#> 500          study   10
#> 519       toxicity    9
#> 313           male    8
#> 192              f    7
#> 197         female    7
#> 303         levels    7
#> 464              s    7
#> 173 environmental    6
#> 180       estrogen    6
```

and then plotting a wordcloud with the function `wordcloud2()` from the package `"wordcloud2"`:

```
wordcloud2(dat_words)
```