Journal of
open research software

Matter, U 2018 RWebData: A High-Level Interface to the Programmable Web. *Journal of Open Research Software,* 6: 11. DOI: https://doi.org/10.5334/jors.201

SOFTWARE METAPAPER

RWebData: A High-Level Interface to the Programmable Web

Ulrich Matter

University of St. Gallen, SEPS-HSG/SIAW, Bodanstr. 8, 9000 St. Gallen, CH
ulrich.matter@unisg.ch

The rise of the programmable web offers new opportunities for the empirically driven sciences. The access to, compilation and preparation of data from the programmable web for statistical analysis can, however, involve substantial up-front costs for the practical researcher. The *R*-package **RWebData** provides a high-level framework that allows data to be easily collected from the programmable web in a format that can be used directly for statistical analysis in *R* without bothering about the data's initial format and nesting structure. It was developed specifically for users who have no experience with web technologies and merely use *R* as statistics software. This paper provides an overview of the high-level functions, explains the basic architecture of the package, illustrates the implemented data mapping algorithm, and discusses **RWebData**'s further development and reuse potential.

Keywords: R; programmable web; web api; rest
Funding statement: The author acknowledges financial support from the University of Basel Research Fund as well as support from the Swiss National Science Foundation (grant 168848).

(1) Overview

Introduction

Digital data from the Internet has in many ways become part of our daily lives. Broadband facilities, the separation of data and design, as well as a broader adoption of certain web technology standards increasingly facilitate the integration of data across different software applications and hardware devices over the web. The Internet is increasingly becoming a programmable web,[1] where data is published not only in HTML-based websites for human readers, but also in standardized machine-readable formats to be "shared and reused across application, enterprise, and community boundaries" [28]. The technical ecology of this programmable web consists essentially of web servers providing data in formats such as Extensible Markup Language (XML) via Application Programming Interfaces (APIs)[2] to other server- or client-side applications (see, e.g., [11] for a detailed introduction to web technologies for *R*-programmers). In the conceptual framework of this paper, APIs serve a dual function: they are the central nodes between different web applications and, at the same time, the central access points for empirical researchers when they want to systematically collect and analyze data from the programmable web. More and more of these access points are becoming available every day. Moreover, this trend is likely to continue with the increasing number of devices to access the Internet and the increasing amount of data recorded by embedded systems (i.e., sensors and applications in devices such as portable music players or cars that automatically feed data to web services).[3]

Background and related packages

While the advantages of accessing the programmable web and integrating it into research projects are very promising, the practical utilization of this technology comes at a cost and demands that researchers possess a specific skill set and a certain knowledge of web technologies. **RWebData** substantially reduces these costs by providing several high-level functions that facilitate the exploration and systematic collection of data from APIs based on a Representational State Transfer (REST) architecture.[4] Moreover, the package contains a unified framework to summarize, visualize, and convert nested/tree-structured web data that works independently of the data's initial format (XML/RSS, JSON, YAML). The package is aimed at empirical researchers using *R* as their daily data analysis- and statistics tool, but who do not have a background in computer science or data science. In addition, several lower level functions of **RWebData** might also be useful for more advanced *R*-programmers who wish to develop client applications to interact with REST APIs. The package thus bridges the gap between the fundamental and very valuable *R*-packages that integrate several web technologies into the *R*-environment (see, e.g., [26, 24,

1, 14]) and the statistical analysis of the programmable web universe.

Interacting with a REST API from within the *R*-environment, therefore, means, sending HTTP requests to a web server and handling the server's response with *R*. In addition to the basic functionality necessary for this, which is delivered in the **base**-package [16], more detailed functions for HTTP requests are provided in packages such as **RCurl** [24], **curl** [13], and **httr** [30]. **RWebData** builds on the **libcurl**-based **RCurl** package, which is designed to interact with web servers hosting a REST API. The implementation is focused on a robust download of resources that checks the received HTTP response for potential problems (e.g., unexpected binary content of the HTTP responses' body-entity) to make sure that the user does not have to specify anything other than the URL.

Most functions related to the fitting and testing of statistical models as well as to exploratory data analysis and data visualization in the *R* computing environment work on data represented in data-frames (a table-like flat representation in which each row represents a data-record/observation and each column a variable describing these records).[5] Data-frames (and their newer variants, data-tables [4] and tibbles [8]) are likely the most common *R*-objects used by researchers when conducting an empirical analysis with *R*. Many input/output-functionalities in *R* serve to import (export) data into (from) *R* as data-frames.[6] However, when it comes to reading data from web APIs into *R*, this is not necessarily the case. The reason lies in the nature of web data formats that allow for more flexibility than solely table-like data representation and come (in the context of web APIs) typically in a nested structure. While a conversion from one table in, e.g., an XML-document to a data-frame is straightforward (see, i.e., `xmlToDataFrame()` in Temple Lang [26]; see also [21]), the conversion of a more complex XML-document with a nested data structure to a flat table-like data representation in *R* or any other computing environment is ex ante less clear and depends on the nature of the data and the purpose the data is being converted for.[7] This is particularly the case for data provided by a non-scientific API that is explicitly made for web developers in order to integrate the data in dynamic websites or mobile applications and not for researchers to integrate the data in their analysis. Not surprisingly, there are different solutions offered in the *R*-environment to map web data formats such as JSON and XML to *R*-objects.[8] Most packages represent web data as nested lists containing objects of different classes such as atomic vectors or data-frames. After converting the original web data to *R*-objects, the user, therefore, often has to extract and rearrange the data records of interest in order to obtain a data representation for statistical analysis. This process becomes even more complex, as queries to the same API method might provide slightly different results depending on the amount of detail embedded in the data records. In these cases, data extraction can become particularly costly if the data set that needs to be compiled depends on many API requests involving various API methods.

Web data conversion

Web data provided via REST APIs are typically in a format such as XML, JSON, or YAML and are not structured in tables containing data values in rows and columns, but are rather organized in a nested (tree-like) structure. However, independent of the data format and data structure, documents provided by a particular REST API have basic aspects in common. Based on these commonalities, this section presents a conceptual and terminological framework as well as a description of how this framework is used to develop the data mapping strategy applied in **RWebData** and serves as the foundation of the data mapping algorithm that will be outlined later on.

Original data structure and data semantics

From the researcher's point of view, the data sets provided by web APIs obey, independent of the raw data format and nesting structure, some very basic data semantics. For these data semantics, I use mainly the same terminology that is nicely outlined by Wickham [31, p. 3]:

1. "A dataset is a collection of *values*, usually either numbers (if quantitative) or strings (if qualitative)."
2. "Every value belongs to a *variable* and an *observation*."
3. "A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units."
4. "An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes."
5. In addition, each observation and each variable belongs to a specific observation *type*. Thus, in a multi-dimensional data set describing, for example, a firm, *observation types* could be employees or shareholders.

The following fictional XML example further illustrates the basic data semantics outlined above.

```
<?xml version="1.0" encoding="UTF-8"?>
<firm>
    <employees>
        <employee>
            <firstName>John</firstName>
            <secondName>Smith</secondName>
        </employee>
        <employee>
            <firstName>Peter</firstName>
            <secondName>Pan</secondName>
        </employee>
    </employees>
    <shareholders>
    <shareholder>
        <ID>S1</ID>
        <Name>Karl Marx</Name>
    </shareholder>
    <shareholder>
        <ID>S2</ID>
        <Name>Bill Gates</Name>
    </shareholder>
    </shareholders>
    <firmName>MicroCapital Ltd</firmName>
    <firmID>123</firmID>
</firm>
```

The XML document contains a data set describing a firm. The variables "firstName" and "secondName" describe observations of type "employee", while observations of type "shareholder" are described by the variables "ID" and "Name". Finally, the variables "firmName" and "firmID" describe the firm itself, which forms another type of observation. The following subsection illustrates how the data would be mapped according to the procedure implemented in **RWebData**.

Mapping nested web data to data-frames

The core idea behind the data mapping procedure in **RWebData** is built around the basic semantics outlined above. According to this system, documents returned from APIs can contain data describing observational units of one type or several different types. **RWebData** returns one data-frame for each observation type. Observations and variables belonging to different types are, thus, collected in different data-frames, each describing one type of observational unit. Some observations might describe the document (or main type of observational unit) itself (i.e., metadata). These are collected in a separate data-frame. **Table 1** (a–c) present the XML-document from the example above after mapping to data-frames according to the outlined mapping procedure. The next subsection explains how this process is implemented in **RWebData**.

Implementation and architecture

Parsing and generic mapping of different web data formats

In order to map nested web data to data-frames, **RWebData** applies, in a first step, existing parsers for different formats (mime-types) of the data in the body of the HTTP response from the API to read the data into *R*.[9] Independent of the initial format, the data is parsed and coerced into a (nested) list representing the tree-structure of the raw data. In a next step, the data is mapped to one or several data-frames depending on the nesting structure and the recurrence of observation types and variables. The data-frames are returned to the user either directly (in a list) or as part of an `apiresponse`-object.

The core idea behind the generic approach to web data conversion in the **RWebData** package is thus to disentangle the parsing of web data from the mapping of the data to data-frames. This allows the parsers for different web data formats to be relatively simple, while the same mapping algorithm is focused on the data semantics and can be applied independent of the initial raw data format. In addition, it allows different parsers' advantages to be combined in order to make the parsing process more robust. The suggested procedure goes hand in hand with the object-oriented approach applied in **RWebData** and provides summary and plot methods that work independent of the initial data format. The modular design of mapping web data to data-frames facilitates the extension of **RWebData**'s compatibility with additional web data formats or alternative parsers that are, for example, optimized for vast web documents. Parsers simply need to read the web data as (nested) lists or in a format that conserves the tree-structure of the raw web data and can be easily coerced into a (nested) list.

Table 1: The same data as in the XML code example above, but mapped to data-frames according to the generic **RWebData** mapping procedure. While (a) contains data describing the firm (the main observational unit) itself, (b) and (c) contain the observations and variables describing the observation types employee and shareholder, respectively.

	firmName	**firmID**
1	MicroCapital Ltd	123

(a) Firm type

	firstName	**secondName**
1	John	Smith
2	Peter	Pan

(b) [Employee type]

	ID	**Name**
1	S1	Karl Marx
2	S2	Bill Gates

(c) [Shareholder type]

Basic data mapping algorithm

Once a document is downloaded from an API and coerced into a nested list, the algorithm consists essentially of two procedures:

1. The extraction of different observation types (and their respective observations and variables) as sub-trees. While reversely traversing the whole data-tree, the algorithm checks at each level (node) of the tree whether either the whole current sub-tree or one of its siblings can be considered an observation type. If so, the respective sub-tree is extracted and saved in a deposit variable. If not, the algorithm traverses further down in the tree-structure. Checks for observation types are defined as a set of control statements that incorporate the above outlined data semantics. Essentially, the recurrence of the same variables as siblings (containing the actual data values as leaf elements) is decisive in order to recognize observation types. The result of this step is a set of sub-trees, each sub-tree containing one observation type and its respective observations and variables.
2. For each of the resulting observation types (in the form of sub-trees), the respective observations are extracted as character vectors and bound as rows in a data-frame, while the variable names are conserved and added as column names. This procedure is built to be robust to observations described by a differing number of variables. The result of this step is a set of data-frames, each describing one observation type and containing individual observations as rows and variables as columns.

Singly occurring leaf nodes that are not nested within one of the detected observation types are collected along the way and then returned in one data-frame. According to the logic of the data semantics and the algorithm outlined above, these remaining data can be seen as metadata

describing the data set as a whole. The Appendix presents a detailed, more technical outline of the data mapping algorithm.

Architecture

RWebData is specifically written to give practical researchers a high-level interface to compile data from the programmable web. The common user will thus normally not be confronted with the range of internal processes outlined above. The work-flow with **RWebData** for the average user thus mainly involves the specification of what data should be requested from what API (in the form of either a list, a data-frame or simply a string with a URL) to one of **RWebData**'s high-level functions in order to obtain the desired data mapped to data-frames. **Figure 1** illustrates the basic architecture of **RWebData**, summarizing its internal functionality and the processing procedure when the package is used by the average user.

RWebData's high level functions take either lists, data frames, or a URL (as a character string) as input values and return a list of data-frames or an apidata-object. First, **RWebData** generates an `apirequest-object`, based on which the HTTP GET and responses are handled. Upon successful interaction with the API, an `apiresponse-object`, containing, inter alia, the raw web data, is generated. Subject to consideration of the mime-type of the raw data, the data is then preprocessed, parsed, and coerced to a nested list. Finally, the data mapping algorithm is applied in order to extract the data in the form of data frames as outlined in the previous section. The latter builds the main part of the package. The procedure is straightforward and user-friendly in practice.

Examples

In order to describe the basic usage of **RWebData**, this section offers a step-by-step introduction to the high-level functionality of the package. Some of the functions described here offer more options, which are not all discussed in this section.[10] The latest version of the package can be directly installed from the GitHub repository via `devtools::install_github()` as shown below.

```
R> # install.packages("devtools")
R> library(devtools)
R> install_github("umatter/rwebdata")
R> # load RWebData
R> library(RWebData)
```

RWebData's implementation is motivated by the convention over configuration paradigm and thus provides a way for the user to interact with APIs using as few specifications as necessary to access the data. **RWebData** then converts data in the most common formats provided by APIs to one data-frame or a list of several data-frames. Hence, the user does not need to specify any HTTP options or understand what XML or JSON is and is able to obtain the data directly in the expected format for statistical analysis. There are primarily two high-level functions in **RWebData** that provide this functionality in different ways: `getTabularData()` and `apiData()`. The following subsection discusses how these functions work and how they can be applied in practice.

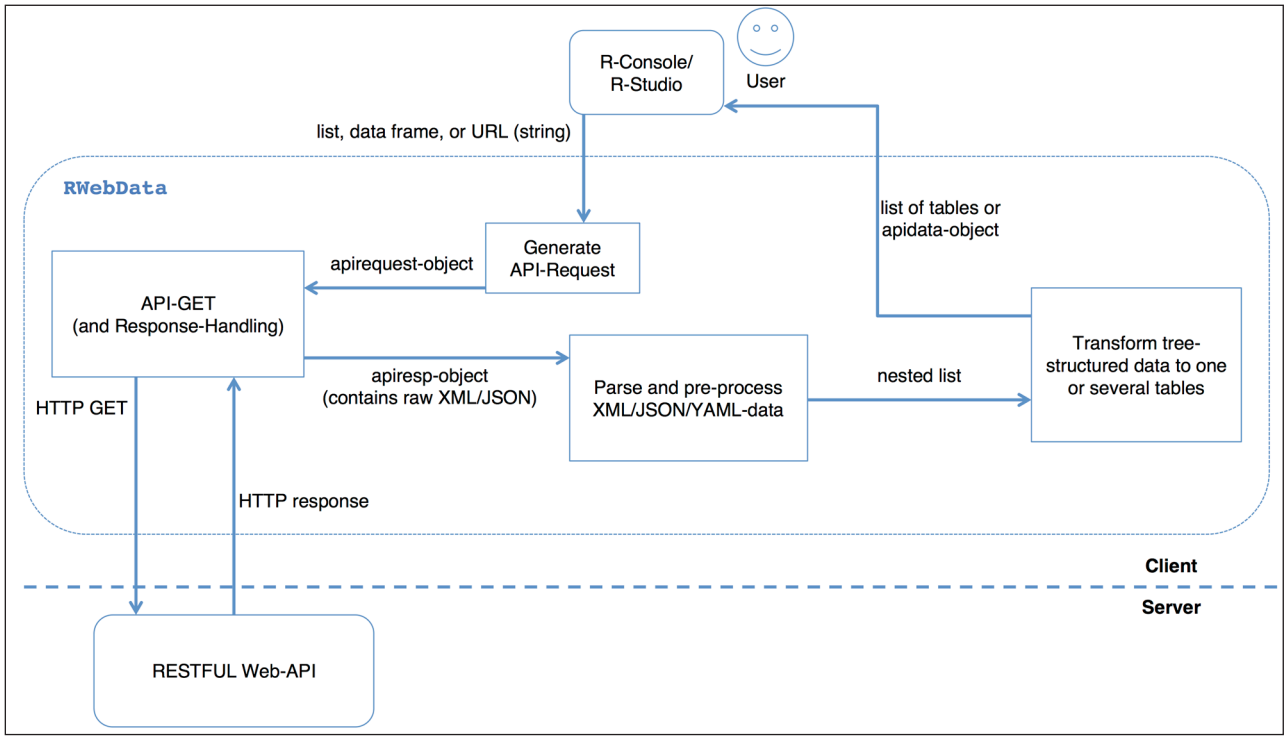

Figure 1: Basic architecture of the RWebData package.

Fetching data from REST APIs

Querying data from a REST API is based on the same client-server principle as opening a website in a web browser. In order to visit a certain website, we normally type a website's address in the web browser's address bar. The browser (a type of web *client*) then sends a message to the web server behind that address, requesting a copy of the specific webpage and then parses and renders the returned document (usually an HTML document) in the browser window. Any website address consists of several components that specify the request to the server. For example, the address `https://www.admin.ch/gov/en/start.html` contains the components `https` (the scheme), `www.admin.ch` (the host name/domain name with the top-level domain `ch`), and the path to the specific webpage (here, an HTML document): `gov/en/start.html`.

In REST terminology, the website's address is a *URL* and the website which the URL locates is a *resource*. The document that the server sends in response to the client's request is a *representation* of that resource. Importantly, every URL points to only one resource and every resource should have only one URL (this is one of the REST principles and is referred to as *addressability*). The message that the browser sends to the web server is a HTTP *GET* request, a HTTP method that essentially requests a representation of a given resource.

REST APIs work basically the same way as the website outlined above. The crucial difference is that their design is intended for programmable clients (rather than humans using a web browser) and that they therefore consist of URLs pointing to resources that are not optimized for graphical rendering, but which instead contain the raw data (often in a format such as XML or JSON that facilitates the integration of the data in a HTML document). However, a URL pointing to an API basically follows the same logic and consists of the same components, including scheme, host name, path, and potentially additional parameters (key-value pairs separated by `&`). Thus, in order to query data from a REST API, we first have to construct the respective URL pointing to the data of interest and then apply the functions provided **RWebData** to query the data. The example below further illustrates this point.

The function `getTabularData()` provides a straightforward way to get data from web APIs as data-frames. In the simplest case, the function takes a string containing the URL to the respective API resource as an input.[11] The function then handles the HTTP request and response, parses the body of the response (depending on the mime-type) and automatically extracts the data records from the nested data structure as data-frames. This enables us to fetch data from different APIs providing data in different formats with essentially the same command.

Consider, for example, the World Bank Indicators API which provides time series data on financial indicators of different countries.[12] We want to use data from that API to investigate how the United States' public debt was affected by the financial crisis in 2008. All we need in order to download and extract the data is the URL to the respective resource on the API.[13]

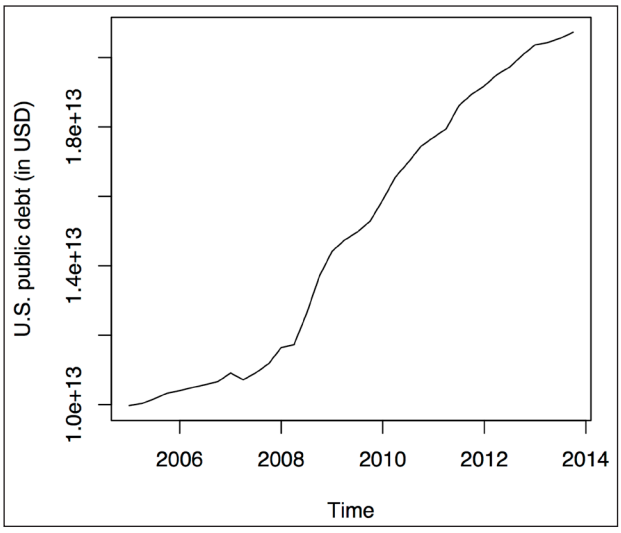

Figure 2: Plot of the time series on United States' public debt extracted from the World Bank Indicators API.

```
R> u <- paste0("http://api.worldbank.org/v2/
countries/USA/indicators", # address
+       "/DP.DOD.DECN.CR.GG.CD?", # query method
+       "&date=2005Q1:2013Q4") # parameters
R> usdebt <- getTabularData(u)
```

Without bothering about the initial format,[14] the returned data is already in the form of a data-frame and is ready to be analyzed (e.g., by plotting the time series as presented in **Figure 2**):

```
R> require(zoo)
R> plot(as.ts(zoo(usdebt$value, as.yearqtr
(usdebt$date))),
     ylab="U.S. public debt(in USD)")
```

The underlying data structure is relatively simple in this example. The World Bank Indicators API is specifically devised to provide data for statistical analysis. The next section considers examples where the provided data cannot easily be thought of as one single table.

Nested data structures

The high-level function `getTabularData()` automatically handles nested data and converts them to a list of various data-frames. It does not, however, provide any information on the initial nesting structure and can only handle individual requests. Alternatively, we can use `apiData()` in order to exploit **RWebData**'s internal classes and methods to handle requests to APIs. The simplest way to call `apiData()` is again with a URL-string pointing to an API resource. `apiData()` returns an object of class `apiresponse`. Such `apiresponse` objects contain additional information about the executed request to the API and support generic plot and summary functions illustrating the structure and content of the retrieved web document. The following example demonstrates the summary methods for `apiresponse` objects with data from the Ergast API (http://ergast.com/mrd/) on Formula 1 race results.

```
R> f1 <- apiData("http://ergast.com/api/
   f1/2013/1/results.json",
+      shortnames=TRUE)
R> summary(f1)
API data summary:
=================

The API data has been split into the following 2
data frames:
         Length Class       Mode
metadata 19     data.frame  list
Results 27      data.frame  list

The respective data frame(s) contain the
following variables:

1. metadata:
xmlns, series, url, limit, offset, total,
season, round, 1, raceName, circuitId, 2,
circuitName, lat, long, locality,
country, date, time,

2. Results:
number, position, positionText, points, driverId,
permanentNumber, code, url, givenName,
familyName, dateOfBirth,
nationality, constructorId, url, name, nationality, grid,
laps, status, millis, time, rank, lap, time,
units, speed,
path,
```

The summary method called by the generic `summary()`, provides an overview of the variables included in the data and shows how **RWebData** has split the data into several data-frames. This is particularly helpful in an early phase of a research project when exploring an API for the first time.

The next example demonstrates the visualization of nested data returned from the Open States API.[15] We query data on a legislator in the Council of the District of Columbia with `apiData()` and call the `generic plot()` on the returned `apiresponse` object. **Figure 3** shows the result of the plot command below. Note that the Open States API is free to use for registered users. Upon registration an api-key is issued that is then added as a parameter-value to each API request (indicated with "`[YOUR-API-KEY]`" in the example below).

```
R> url <- "http://openstates.org/api/v1/
legislators/DCL000004/
+  ?apikey=[YOUR-API-KEY]"
R> p <- apiData(url)
R> plot(p, type="vertical")
```

The `apiresponse` plot method illustrates the nesting structure of the original data by drawing entities (variables or types) as nodes and nesting relationships as edges in a Reingold-Tilford tree-graph [18] (see also [3] for the implementation in *R* on which **RWebData** relies). The option `type="vertical"` displays the node labels vertically in order to make long node names better readable in the graph.

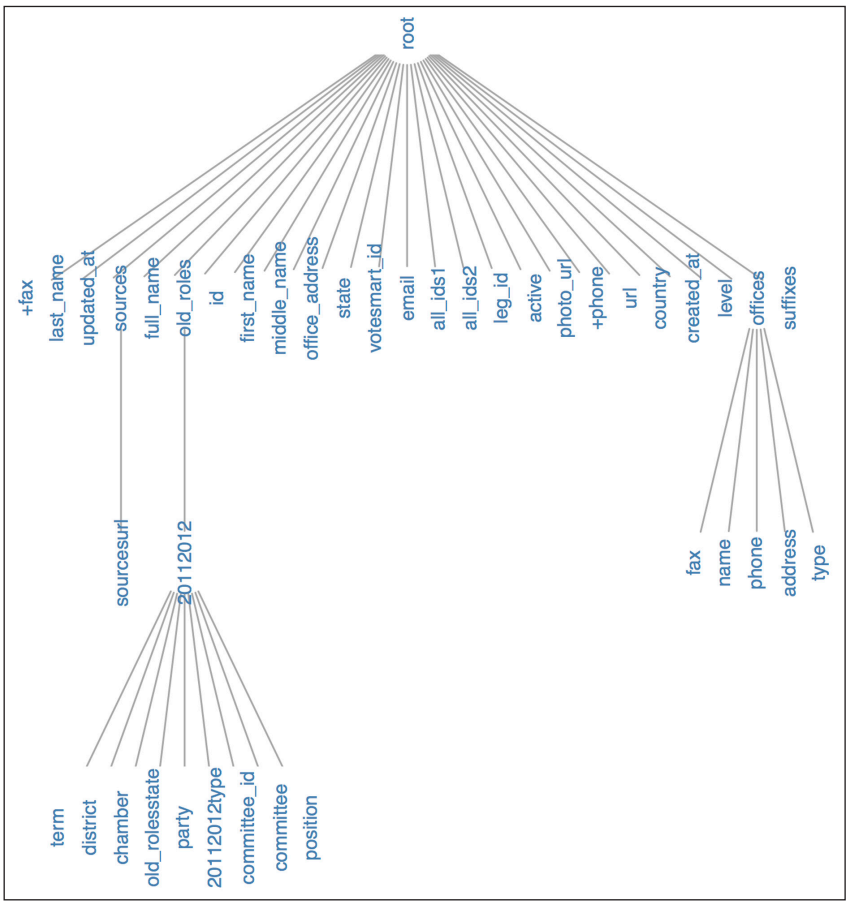

Figure 3: Reingold-Tilford tree-graph illustrating the nested structure of the raw JSON data from the Open States API.

The transformed data (in the form of a list of data-frames) saved in an `apiresponse-object` can then be accessed with `getdata ()`. In the current example, the data has been split into two data-frames: one with meta data containing general variables describing the legislator, and one containing data on the legislator's roles in the 2011–2012 session.

```
R> pdata <- getdata(p)
R> summary(pdata)

           Length  Class       Mode
metadata   22      data.frame  list
2011-2012 11      data.frame  list
```

Interactive sessions

So far, we have only considered individual requests to APIs. In practice, users might want to query an API in many ways in an interactive session to explore the data and assemble the data set that meets their needs. This can easily be done with **RWebData**. We continue with the Open States API example. First, we fetch general data on all legislators in the DC Council.

```
R> url <- "http://openstates.org/api/v1/legislat
ors/?state=dc&chamber=upper
+   &apikey=[YOUR-API-KEY]"
R> dc_council <- getTabularData(url)
```

The goal is to combine these general data with data on the legislators' roles. In particular, we want to download the detailed role data on all legislators and combine these within one data set. As there is no API method provided to obtain these data with one call to the API, we use `apiDownload()` to handle all individual requests at once. By inspecting the `dc_council` data-frame from above, we see that the fifth column of that data-frame contains all the IDs (id) pointing to the respective resources on individual council members. We simply paste these ids together with the `legislators`-method and use the resulting URLs as the function argument to `apiDownload()` in order to obtain all the data compiled in one data-frame.

```
R> head(names(dc_council))
[1] "fax" "last_name" "updated_at" "full_name" "id"
[6] "first_name"
R> api_requests <- paste0("http://openstates.
org/api/v1/legislators/",
+             dc_councilid,
+             "/?apikey=",
+             "YOUR-API-KEY")
R> dc_leg_roles <- apiDownload(api_requests)
```

During the download, `apiDownload()` indicates the number of queries processed on a progress bar printed to the R console. Moreover, `apiDownload()` periodically saves the processed data locally in a temporary file. This relieves the working memory assigned to the *R*-session and makes large downloads more robust to network interruptions. A summary of the resulting `dc_leg_roles` is presented in the Appendix.

Outlook: Writing interfaces to REST APIs

In the context of an ongoing research project, it might be more comfortable to have a specific function for specific queries to the same API. The further development of **RWebData** is aimed at providing a way to write such functions automatically. The current implementation of this functionality is still being tested and made more robust.[16]

The following example illustrates how this newest extension of the package can be applied. Given the parameters and the base URL for the respective API method/resource, `generateQueryFunction()` writes a function that includes all the functionality of the package to handle queries to that specific API method/resource. The same works for APIs that accept requests via form URLs, such as the MediaWiki API.[17] Here, only the base URL (i.e., the *endpoint* of the API) and the respective parameters have to be specified.

```
R> endpoint <- "https://en.wikipedia.org/w/api.php?"
R> # set parameters (set to NA if no default
   value in query-function wanted)
R> query_params <- data.frame
   (parameters=c("action", "format", "titles",
   "prop"),
+           values = c("query", "xml", NA, NA))
R> mediaWiki <- generateQueryFunction(x=query_
   params, base.url=endpoint)
R>
```

The new function `mediaWiki()` can now be used to query the MediaWiki API for information about Wikipedia pages.

```
R> # get meta data on the Wikipedia page about
   the Internet
R> page_info <- mediaWiki(titles = "Internet",
   prop = "info")
R> page_info[,1:5]

  page._idx page.pageid page.ns page.title page.
contentmodel
1   14539     14539     0 Internet      wikitext

R> # get information about the latest revision
of the same Wikipedia page
R> revisions <- mediaWiki(titles = "Internet",
prop = "revisions")
R> revisions[,1:5]

  rev.revid rev.parentid rev.minor  rev.user
rev.timestamp
1 823121130    822163533           Ragityman
2018-01-30T10:58:38Z
```

The MediaWiki API query method allows batch queries of several Wikipedia page titles (separated with the pipe operator '|'). `mediaWiki()` automatically recognizes the different observations of the same observation-type and returns the data as a data-frame with one row per observation:

```
R> page_info <- mediaWiki
   (titles = "USA|France|Germany|Switzerland",
+   prop = "revisions")
R> dim(page_info)
[1] 4 14

R>
```

With only three lines of code, we have generated a basic MediaWiki API *R*-client library that automatically provides the web data as data-frames.

Discussion

Empirically driven research can benefit substantially from the rapidly growing programmable web as a data source covering countless dimensions of socio-economic activity. However, the purpose and goals that drove the initial development of web APIs focused on providing general data for public use via dynamic websites and other applications, and not on the provision of data formatted for scientific research and statistical analyses. This leads to technical hurdles a researcher has to overcome in order to compile such data in a format that is suitable for statistical analysis. The *R* package **RWebData** presented here suggests a simple high-level interface that helps to overcome such technical hurdles (and the respective costs) associated with the statistical analysis of data from the programmable web. The package contributes to a frictionless and well documented raw data compilation and data preparation process that can help to increase the replicability and reproducibility of original research based on data from the programmable web. As pointed out in [2], empirical research with newly available digital data from novel sources generally poses new challenges to the reuse of data as well as the reproduction of results. Empirical research based on newly available public data from web sources is a challenge demanding scientific rigor. With the further development of **RWebData**, an *R*-script documenting how the package's high-level functions have been applied to compile data as well as any further step in the data preparation and analysis are enough to ensure the replicability of a study based on big public data from the programmable web.

Appendix

Data mapping algorithm

RWebData parses and coerces the raw web data to a nested list representing the tree-structure of the data. Call this list $\vec{x}$.

The data mapping algorithm consists of two main parts. First, $\vec{x}$ is split into n lists representing sub-trees, one for each *observation type* in the data. The key problem that the algorithm has to solve at this step is the identification of cutting points (i.e., what part of the tree belongs to what observation type). Second, each resulting sub-tree is then split into individual character vectors. One for each *observation*. The individual observations are then stacked together in one data-frame with each vector (observation) as a row. Algorithm 1 presents a formal description of these procedures. In the resulting data-frames, each row *i represents one observation*, and each column *j represents a variable/characteristic* of the *n* observations. The data-frames are then returned in a list.

In order to make the formal descriptions of the procedures in the data mapping algorithm conveniently readable, the pseudo-code describes simple versions of these procedures (that are not necessarily most efficient). The algorithms' *R*-implementations in **RWebData** are more efficient and contain more control statements to ensure robustness. The actual *R*-implementation relies partly on existing *R* functions and favors vectorization over for-loops in some cases.

Algorithm 1 Data mapping algorithm

```
1: procedure TYPES (x⃗)
2:   Types ← empty list
3:   deposit ← empty list
4:   if x⃗ is an observation type then
5:     add x⃗ to Types
6:     return Types
7:   end if
8:   if x⃗ contains a part i at the highest nesting level that
     is an observation type then
9:     add i to Types
10:    remove i from x⃗
11:  end if
12:  for all elements i in x⃗
13:    if i is a non-empty list then
14:      if i is an observation type then
15:        add i to Types
16:      else
17:      apply this very procedure to i      ▷ recursive call
18:      add the resulting observation types to Types
19:      add the remaining leaves (metadata) to deposit
20:      end if
21:    else
22:    add i to deposit                      ▷ it's a leaf node
       (i.e., just a value)
23:    end if
24:  end for
25: end procedure
26: procedure OBSERVATIONS (Types)
27:    rows ← empty list
28:    for all obs_type in Types
29:      for all observation in obs_type do
30:        unlist and transpose observation
           ▷ extract leaves as vector
31:        add resulting vector to rows
32:      end for
33:      bind rows to one data-frame (preserving
         variable names)
34:    end for
35: end procedure
```

Open States API

```
R> url <- "http://openstates.org/api/v1/legislators/?state=dc&chamber=upper
+  &apikey=[YOUR-API-KEY]"
R> c_council <- getTabularData(url)
R> api_requests <- paste0("http://openstates.org/api/v1/legislators/",
                      dc_council$id,
                      "&apikey=",
                      "YOUR-API-KEY")

  |
  |                                                      |   0%
  |
  |==============================                        |  50%
  |
  |============================================================| 100%

R> # explore the data
R> summary(dc_leg_roles)

          Length Class      Mode
metadata  55     data.frame list
2013-2014 13     data.frame list
2011-2012 13     data.frame list
offices    7     data.frame list
```

Quality control

The data mapping algorithm has been tested in unit tests with various formats (see https://github.com/umatter/RWebData/blob/master/testing.Rmd as well as https://github.com/umatter/RWebData/blob/master/unit_testing.R for details on the tests and https://github.com/umatter/RWebData/blob/master/testing.html for the test results). **RWebData** is delivered with a set of documents with hierarchical data structures in different formats on which the package's own examples are based:

```
R> example(XMLtoDataFrame)
R> example(YAMLtoDataFrame)
R> example(JSONtoDataFrame)
```

In addition to unit testing, the code has already been tested extensively in several real world applications (both on Mac OSX and Ubuntu), both in the course of the author's own ongoing research projects in the areas of political economics and political science [6] as well as with regard to various openly accessible web APIs. A list of these APIs is provided and further maintained and extended in the code repository of **RWebData** (see https://github.com/umatter/RWebData/blob/master/testing.Rmd).

(2) Availability

Operating system

RWebData is compatible with any operating system that is compatible with R 3.2.3.

Programming language

RWebdata is written in and for R (>= 3.2.3).

Additional system requirements

There are no additional system requirements apart from (for most parts of RWebData) a working Internet connection.

Dependencies

igraph (>= 1.0.1), RCurl (>= 1.95–4.8), jsonlite (>= 1.3), XML (>= 3.98–1.5), plyr (>= 1.8.4), stringr (>= 1.2.0), XML2R (>= 0.0.6), httr (>= 1.2.1), mime (>= 0.5), yaml (>= 2.1.14), RJSONIO (>= 1.3–0), methods (>= 3.2.3), (devtools to install the package from GitHub)

List of contributors

Ulrich Matter: University of St. Gallen, SEPS-HSG/SIAW, Bodanstr. 8, 9000 St. Gallen, Switzerland, ulrich.matter@unisg.ch. Ingmar Schlecht: Faculty of Business and Economics, University of Basel, Peter Merian-Weg 6, 4002 Basel, Switzerland.

Software location

Archive

Name: GitHub

Persistent identifier: https://doi.org/10.5281/zenodo.1161954

Licence: GPL-2, GPL-3

Publisher: Ulrich Matter
Version published: 0.2.1
Date published: 01/29/2018

Code repository
Name: GitHub
Persistent identifier: https://github.com/umatter/RWebData
Licence: GPL-2, GPL-3
Date published: 12/02/2015

Language
R 3.2.3

(3) Reuse potential

RWebData (both in the current and previous versions) has already been used in the course of several research projects (see, for example, Matter and Stutzer [6]). With the increasing expansion of the programmable web and thus the amount of digital data covering various aspects of social, political, and economic processes, the reuse potential for **RWebData** is expected to be high, especially in disciplines where *R* is widely used for statistical analysis and data preparation such as economics and political science.

RWebData is explicitly made for researchers without a background in web technologies/programming and who are predominantly using *R* as statistics software for their every-day empirical research. Together with the increasing relevance of digital data from the web for empirical research in various disciplines and fields, **RWebData** has a broad and growing potential user base. In addition, the publication of this software (particularly the extensions mentioned in the outlook) provides opportunities for other researchers to build on the code provided in **RWebData** in order to write their own custom made API clients for data collection.

Notes

[1] I use the term programmable web synonymously with "Semantic Web" or "Web of Data" and as conceptually motivated by [23].

[2] Web APIs are a collection of predefined HTTP requests and response messages used to facilitate the programmatic exchange of data between a web server and web clients.

[3] The number of publicly accessible web APIs grew from around 1,000 at the end of 2008 to over 10,000 at the end of 2013 [15]. Turner et al. [27] estimate that the share of available digital data stemming from embedded systems will rise to 10% by 2020 and will include up to 32 billion devices connected to the Internet. See also [5] for a discussion of how this 'Internet of things' could be expanded on a crowd-sourced basis and employed for big-data analytics.

[4] See [19] for an introduction to REST APIs.

[5] Technically, a data-frame can contain other *R*-objects than just scalars (i.e., another data-frame). However, in the vast majority of applications in the context of statistical analysis and data visualization, data-frames are used for a table-like flat data representation.

[6] See, i.e., data from CSV- and similar text files: `read.table()` in [16] or similar functions in [33], Microsoft Excel files: `read.xls()` in [29], data from other statistical computing environments such as Stata: `read.dta()` in [17], or data from ODBC-data bases `sqlQuery()` in [20].

[7] See, e.g., the classical problem of mapping a set of XML-documents to a relational data base scheme (RDBS; i.e., a set of linked tables) with or without knowing the scheme behind the XML-documents [9, 7]. See also, in the *R* context, the different implementations of mapping JSON to *R*-objects in [1], [25], or [14] as well as a detailed discussion of this matter in [12].

[8] See, e.g., the CRAN Task View on web technologies and services (http://cran.r-project.org/web/views/WebTechnologies.html), the CRAN Task View on open data (https://github.com/ropensci/opendata), as well as the rOpenSci (https://ropensci.org/packages/) for an overview of popular *R*-packages that parse data from the web and map it to *R*-objects. Some contributions in this area also focus on the automated information extraction from traditional websites made for human interaction. Such methods are commonly summarized under the terms "web scraping" or "screen scraping". See, e.g., the *R*-package **rvest** [32] for functions that facilitate web scraping as well as [10] for a practical introduction to web scraping with *R*.

[9] In the case of JSON, either the parser provided by Ooms et al. [14] or (in order to increase robustness in special cases) the one contributed by Temple Lang [25] is applied. XML-documents or RSS-documents are parsed with the parser provided in [26] and YAML-documents with the parser provided by Stephens [22].

[10] Details on all these options are provided by the respective *R* help files. Users are generally encouraged to read the detailed package documentation of **RWebData** (https://github.com/umatter/RWebData/blob/master/RWebData.pdf).

[11] The URL to the resource which a user wants to query is very easy to find in any REST API documentation. How to call the respective resource methods of an API with URLs is the essential part of such documentations.

[12] Note that this is an example of an API which is – unlike most APIs – explicitly made for researchers retrieving data from the web. Nevertheless, retrieving data from this API with functions from, e.g., the **XML** package [26] is not necessarily straightforward for users without a background in web technologies.

[13] How to build query-URLs in the specific case of the World Bank Indicators API is well documented on https://datahelpdesk.worldbank.org/knowledgebase/articles/898581-api-basic-call-structure.

[14] The World Bank Indicators API provides time series data by default as XML in a compressed text file. Handling solely one API query of this type might

already involve many steps to fetch and extract the data with the existing lower level functions. And, thus, might be tedious for a user who only has limited experience with web technologies.

[15] See https://sunlightlabs.github.io/openstates-api/ for details.

[16] Note that this additional feature is built on top of the matured features shown in the examples above. The early stage and future development of this particular extension thus does thus not affect the other functionalities of the package that have already been tested extensively.

[17] See https://en.wikipedia.org/w/api.php? for a detailed documentation of this API.

Acknowledgements

I am grateful to Dietmar Maringer, Armando Meier, Reto Odermatt, Michaela Slotwinski, Alois Stutzer, as well as seminar participants at the University of Basel and the University of Oxford for helpful remarks. Special thanks go to Ingmar Schlecht for many productive discussions on software development and the methodological aspects of this paper. I also thank Joerg Kalbfuss for excellent research assistance.

Competing Interests

The author has no competing interests to declare.

References

1. **Couture-Beil, A** 2013 rjson: JSON for R. R package version 0.2.13. URL: http://CRAN.R-project.org/package=rjson.
2. **Crosas, M, Honaker, J, King, G** and **Sweeney, L** 2015 ANNALS of the American Academy of Political and Social Science, 659(1): 260–273. DOI: https://doi.org/10.1177/0002716215570847
3. **Csardi, G** and **Nepusz, T** 2006 InterJournal Complex Systems, 1695. URL: http://igraph.org.
4. **Dowle, M** and **Srinivasan, A** 2017 data.table: Extension of 'data.frame'. R package version 1.10.4. URL: https://CRAN.R-project.org/package=data.table.
5. **Helbing, D** and **Pournaras, E** 2015 *Nature*, 527(7576): 33–34. DOI: https://doi.org/10.1038/527033a
6. **Matter, U** and **Stutzer, A** 2016 Does Public Attention Reduce the Influence of Special Interest Groups? Policy Positions on SOPA/PIPA Before and After The Internet Blackout Berkman Klein Center Research Publication 2016–22 Berkman Klein Center for Internet & Society at Harvard University. DOI: https://doi.org/10.2139/ssrn.2884316
7. **Men-hin, Y** and **Fu, A W c** 2001 In: *'Proceedings of the 81h International Workshop on Knowledge Representation Meets Databases (KRDB 2001), Rome, Italy'*.
8. **Müller, K** and **Wickham, H** 2017 tibble: Simple Data Frames. R package version 1.3.4. URL: https://CRAN.R-project.org/package=tibble.
9. **Moh, C H, Lim, E P** and **Ng, W K** 2000 In: *'Proceedings of the fth ACM Conference on Digital libraries'*, 67–76. ACM.
10. **Munzert, S, Rubba, C, Meissner, P** and **Nyhuis, D** 2014 Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining John Wiley & Sons Chichester, UK. DOI: https://doi.org/10.1002/9781118834732
11. **Nolan, D** and **Temple Lang, D** 2014 XML and Web Technologies for Data Sciences with R UseR! Springer: New York. DOI: https://doi.org/10.1007/978-1-4614-7900-0
12. **Ooms, J** 2014 ArXiv e-prints. URL: https://arxiv.org/abs/1403.2805.
13. **Ooms, J** 2017 curl: A Modern and Flexible Web Client for R. R package version 3.0. URL: https://CRAN.R-project.org/package=curl.
14. **Ooms, J, Temple Lang, D** and **Wallace, J** 2014 jsonlite: A smarter JSON encoder/decoder for R. R package version 0.9.8. URL: http://CRAN.R-project.org/package=jsonlite.
15. **ProgrammableWeb** 2014 'Programmableweb research center: Growth in web apis from 2005 to 2013'. URL: www.programmableweb.com/api-research.
16. **R Core Team** 2013 R: A Language and Environment for Statistical Computing R Foundation for Statistical Computing Vienna, Austria. URL: http://www.R-project.org/.
17. **R Core Team** 2014 foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ... R package version 0.8-59. URL: http://CRAN.R-project.org/package=foreign.
18. **Reingold, E M** and **Tilford, J S** 1981 *IEEE Transactions on Software Engineering*, 7(2): 223–228. DOI: https://doi.org/10.1109/TSE.1981.234519
19. **Richardson, L** and **Amundsen, M** 2013 RESTful Web APIs O'Reilly Media Cambridge.
20. **Ripley, B** and **Lapsley, M** 2013 RODBC: ODBC Database Access. R package version 1.3-0. URL: http://CRAN.R-project.org/package=RODBC.
21. **Sievert, C** 2014 XML2R: EasieR XML data collection. R package version 0.0.6. URL: http://CRAN.R-project.org/package=XML2R.
22. **Stephens, J** 2014 yaml: Methods to Convert R Data to YAML and Back. R package version 2.1.11. URL: http://CRAN.R-project.org/package=yaml.
23. **Swartz, A** 2013 In: Hendler, J and Ding, Y (eds.), *'Synthesis Lectures on The Semantic Web: Theory and Technology'*. Morgan & Claypool Publishers.
24. **Temple Lang, D** 2013a RCurl: General network (HTTP/FTP/...) client interface for R. R package version 1.95-4.1. URL: http://CRAN.R-project.org/package=RCurl.
25. **Temple Lang, D** 2013b RJSONIO: Serialize R objects to JSON, JavaScript Object Notation. R package version 1.0-3. URL: http://CRAN.R-project.org/package=RJSONIO.
26. **Temple Lang, D** 2013c XML: Tools for Parsing and Generating XML Within R and S-Plus. R package version 3.95-0.2. URL: http://CRAN.R-project.org/package=XML.
27. **Turner, V, Gantz, J F, Reinsel, D** and **Minton, S** 2014 The digital universe of opportunities: Rich data and the increasing value of the internet of things White paper IDC Framingham, MA.

28. **W3C** 2013 'W3c semantic web activity: What is the semantic web?' URL: www.w3.org/2001/sw/.
29. **Warnes, G R, Bolker, B, Gorjanc, G, Grothendieck, G, Korosec, A, Lumley, T, MacQueen, D, Magnusson, A** and **Rogers, J** and others 2014 gdata: Various R Programming Tools for Data Manipulation. R package version 2.13.3. URL: http://CRAN.R-project.org/package=gdata.
30. **Wickham, H** 2014a httr: Tools for Working with URLs and HTTP. R package version 0.4. URL: http://CRAN.R-project.org/package=httr.
31. **Wickham, H** 2014b *Journal of Statistical Software*, 59(10): 1–23. URL: https://www.jstatsoft.org/index.php/jss/article/view/v059i10/v59i10.pdf. DOI: https://doi.org/10.18637/jss.v059.i10
32. **Wickham, H** 2015 rvest: Easily Harvest (Scrape)Web Pages. R package version 0.2.0. URL: http://CRAN.R-project.org/package=rvest.
33. **Wickham, H** and **Francois, R** 2015 readr: Read Tabular Data. R package version 0.1.0. URL: http://CRAN.R-project.org/package=readr.

How to cite this article: Matter, U 2018 RWebData: A High-Level Interface to the Programmable Web. *Journal of Open Research Software* 6: 11, DOI: https://doi.org/10.5334/jors.201

Submitted: 30 October 2017 **Accepted:** 07 February 2018 **Published:** 21 February 2018

]u[*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press

OPEN ACCESS