How to retrieve sustainable energy Data (solar and wind) from Entso REST API with R

Jeroen Groot

April 16, 2018

Context

A lot of webapplications offer interesting data for research. However collecting their data by hand can be tedious or sometimes just impossible. Many owners of webapplications also offer their data through an API (Application programming interface) for the purpose of offering the (raw) data in bulk or for implementation in another web-application. In this tutorial a method for getting data from a REST API will be shown for a specific case: the Entso-E tranparancy platform.
We will be using dplyr pipes and functions from the apply-familyin combination with rvest html/xml functions so a basic understanding of these in R is recommended before starting this tutorial.

Approach

A RESTful API is a method of allowing communication between a web-based client and a server that employs representational state transfer (REST) constraints. For the purpose of this tutorial we will not be going into the details of RESTful rules, but do add that the return type of RESTful API's is usually structured or semi-structured. This means data in a format like: HTTP, URI, JSON, or XML. The Entso-E REST API uses XML.

The first step is to ask for an API-token from the nice people at Entso-E, do this through an email to `transparency@entsoe.eu`. Be sure to add your email-address in the email.

The following steps will be discussed in the next chapter and paragraphs:

1. Building the url
2. Send GET-request and get response
3. Processing the GET-response
4. Putting the data together

Building the script

Step 0: API-Token

First priority, as mentioned before, is to have the API-token. This gives us access to the API, without it all responses will be denied.

```
token <- "<API-Token>"
```

Secondly the R-libraries used and the reason they are included:

```r
library(httr) # Contains the GET() function to get a response
library(dplyr) # Makes data operations a lot easier and more readable
library(rvest) # For processing the XML returns from the GET-response
```

Step 1: Getting the data we need

The REST API needs to be contacted through a url containing all the infromation it needs to send a response with the data. To do this we need to build up this url ourselves.

The url needs different arguments for different parts of the API. For our purpose of getting sustainable Energy data we need the following:
1. A document Type (e.g. generation or load) 2. A process Type (forecast or realised)
3. A country code 4. Power type (e.g. Solar, Wind on shor, Nuclear)
5. Time interval in the format "%Y-%m-%d T%H:%MZ"

Entso-E offers documentation in tables found here: https://transparency.entsoe.eu/content/static_content/download?path=/Static%20content/web%20api/RestfulAPI_IG.pdf

The first step is to decide what data type, document type, country code and energy type we need. Entso-E offers many different data sets for many types of energy generation or network loads. This can get difficult to read if you don't know what you are looking for.

The following setting is for actual generation (not forecasts) of solar energy in the Netherlands.

```r
documentType <-"&documentType=A75" # documentType = A73: actual generation

processType <- "&processType=A16" # processType = a16: realised

psrType <- "&psrType=B16" # psrType= B16: Solar, B19: Wind Onshore

In_Domain <- "&in_Domain=10YNL----------L" # in_Domain = 10YNL----------L: the Netherlands
```

The next decision to make is what time interval we need. In this tutorial we will be getting the data over 2015. Note that the API has a limit of maximum one year per request and this needs to be after `2015-01-01`.

```r
startdate <- "2015-01-01"
starttime <- "00:00"
enddate <- "2016-01-01"
endtime <- "00:00"
```

The time interval we are going to include in the url needs the be in a specific form.

```r
timeinterval <- paste0("&timeInterval=",
                       startdate,"T", starttime,"Z/",
                       enddate,"T",endtime,"Z")
```

Building the url

Now to build the url we paste the previous values together with the domain of the REST API and the assigned token we retrieved through the email.

```r
geturl <- paste0("https://transparency.entsoe.eu/api?securityToken=",
                 token,
                 documentType, processType, psrType,
                 In_Domain, timeinterval)
```

Step 2: Sending a GET request, GETting a response

```
GET.response <- GET(geturl)
```

Thats it, you thought it was going to be more difficult weren't you?

Off course we need to check if everything came in correctly. To do this we look at the `status_code` in the response.

```
GET.response$status_code
```

```
## [1] 200
```

This needs to be 200, anything else means something went wrong before. A list of status_codes and what they mean can be found at:
http://www.restapitutorial.com/httpstatuscodes.html

Step 3: Processing the response

If everything has gone the way it should so far, we should have ended up with a GET-response with status_code 200 and raw content somewhere in the response.

To turn this into something we can actually analyse using our favorite R-tools we need to process it. The content in the GET-response contains a list of different timeseries broken up into an undefined interval. This makes the next few step a bit more difficult.

The codechunk below does the following: - Translate Raw content into Character
- Read as HTML (XML and HTML sturctures are very similar) - Point to second level children of the returned nodes

```
Children <- GET.response$content %>%
  rawToChar() %>%
  read_html() %>% html_children() %>% html_children()
```

This means we now have pointers towards the level where the timeseries are in the xml. The pointers are used instead of the content because the nodes still have children below them.

We don't know before hand what the lengths of the different timeseries are so we need to get these from the second level children.

Take the starting and the ending time of all timeseries. Note that the timezone is GMT (so no summer/winter time like CET/CEST).

```
Intervals <- NA
Intervals$start <- Children %>%
  html_nodes("timeseries") %>% html_nodes("period") %>%
  html_nodes("timeinterval") %>% html_nodes("start") %>% html_text() %>%
  as.POSIXct(format="%Y-%m-%d T%H:%MZ", tz="GMT")

Intervals$end <- Children %>%
  html_nodes("timeseries") %>% html_nodes("period") %>%
  html_nodes("timeinterval") %>% html_nodes("end") %>% html_text() %>%
  as.POSIXct(format="%Y-%m-%d T%H:%MZ", tz="GMT")
```

Now to extract the actual values of the timeseries. After pointing towards the timeseries nodes we need to usethe sapply functions to get the values because the timeseries have three (!) levels of children below them. These are “period”, “point” and then the “quantity”. The reasing for not doing this into one big list is that the different timeseries might have missing data in between them. After getting the right pointers, read them as text and translate type character into numeric.

```
Timeseries <- Children  %>% html_nodes("timeseries") %>%
      # for all nodes named timeseries point towards quantity
      sapply(., function(timeseries) timeseries %>% html_nodes("period") %>%
               html_nodes("point") %>% html_nodes("quantity")) %>%
      # for all nodes named quantity read their text and set as numeric
      sapply(., function(quantities) quantities %>% html_text() %>% as.numeric())
```

So far: We have taken the information we need from the content by pointing towards the right place inside the response and then taking only the information we need from those locations. So we have now ended up with a list of timeseries values (unlabelled) in numeric and a list of starting and ending timestamps of these timeseries in POSIX.

Step 4: Putting the data together

To get the data together we need to build a timeseries. As we have only the starting and ending timestamps of the timeseries we first need to build sequences out of them. For the Dutch data, the resolution is 15 minutes. This differs for different countries and energy types.

```
Sequence <- mapply(function(x,y) seq(from = x, to = y, by = "15 min"),
                   Intervals$start,
                   Intervals$end - (15*60))
```

Note that `(15*60)` is subtracted from the end of the Interval. The reason for this is that the `seq()` function includes the last value in the sequence. The length of the sequences should now be the same as the length of the timeseries we have.
Check:

```
data.frame(Timeseries.length = sapply(Timeseries, length),
           Sequences.length = sapply(Sequence, length))
```

```
##   Timeseries.length Sequences.length
## 1             11800            11800
## 2              5280             5280
## 3               576              576
## 4              4992             4992
## 5              7008             7008
## 6               192              192
## 7                96               96
## 8              2880             2880
```

Now binding these together as data.frames.

```
Together <- mapply(data.frame, Timestamp = Sequence, Solar = Timeseries, SIMPLIFY=FALSE)
```

Then binding the rows together

```r
Together <- do.call(rbind, Together)
```

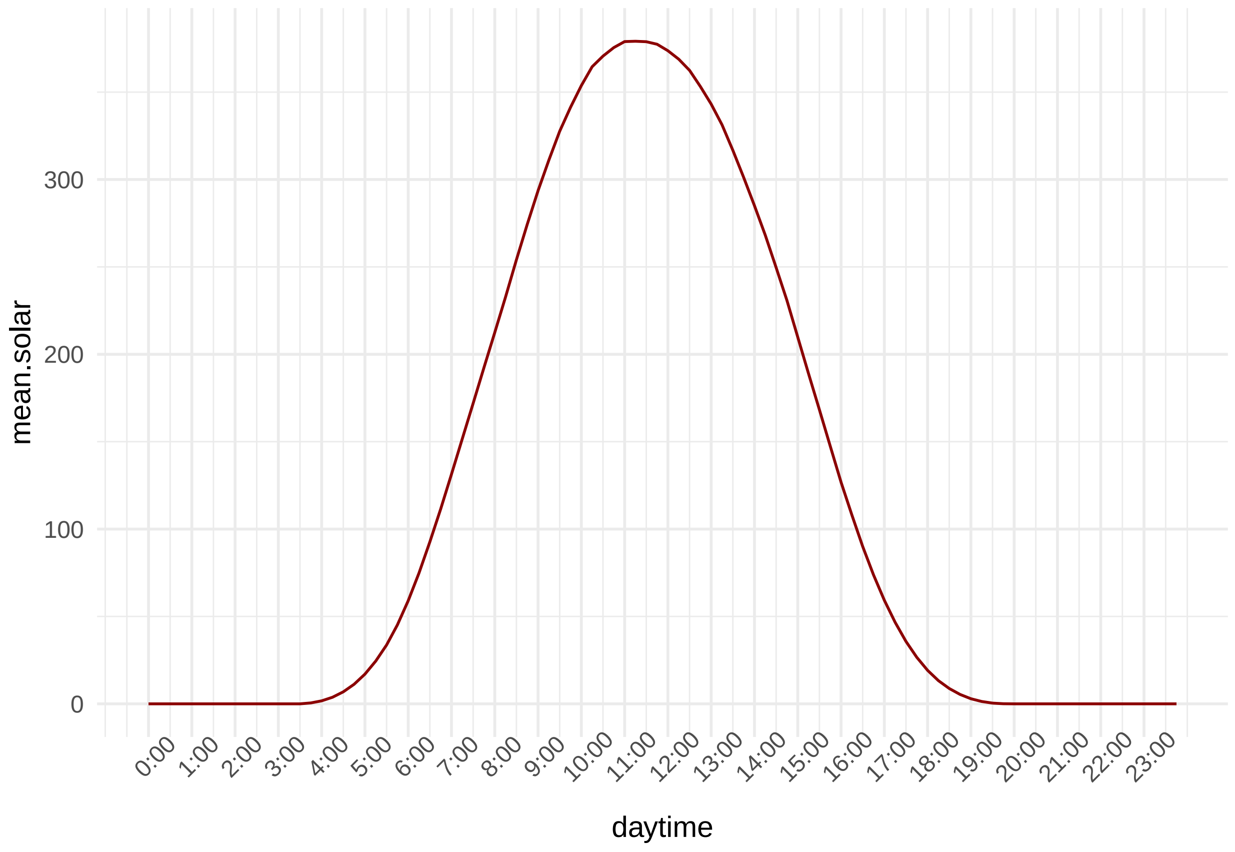
We now ended up with the following: A data frame with every 15 minute values from the Entso-E REST API on Solar generation in the Netherlands.

```
## Observations: 32,824
## Variables: 2
## $ Timestamp <dttm> 2015-01-01 00:00:00, 2015-01-01 00:15:00, 2015-01-0...
## $ Solar     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
```

This gives us the possibility to do analysis and build images like:

```r
library(ggplot2)
library(lubridate)

Together %>% mutate(daytime = hour(Timestamp)+minute(Timestamp)/60) %>%
  group_by(daytime) %>% summarise(mean.solar = mean(Solar, na.rm=TRUE)) %>%
  ggplot()+
    geom_line(aes(x=daytime, y=mean.solar), col = "Darkred")+
    scale_x_continuous(breaks=0:23, labels=paste0(0:23,":00"))+
    labs(list(x="Time of day", y="Average solar generation",
              title="Daily average solar generation",
              subtitle="in the Netherlands 2015"))+
    theme_minimal()+
    theme(axis.text.x = element_text(angle=45))
```


mean.solar
0:00
1:00
2:00
3:00
4:00
5:00
6:00
7:00
8:00
9:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
daytime