

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

## Introduction

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Welcome

---

# Welcome!

---

Thank you for your interest in the course!

I'm glad that I will have the opportunity to give you a gentle introduction to web scraping with R.

Just a few words on my personal background:

- Lecturer in Political Data Science at the Hertie School of Governance
- political scientist by training
- working with web-based data since about 2010
- what I do with web data: measure public awareness, news consumption, political behavior

## **Goals and outline**

---

# Goals

---

After attending this course, ...

- you have acquired basic knowledge of web technologies
- you are able to scrape information from static and dynamic websites using R
- you are able to access web services (APIs) with R
- you can build up and maintain your own original sets of web-based data

# Course outline

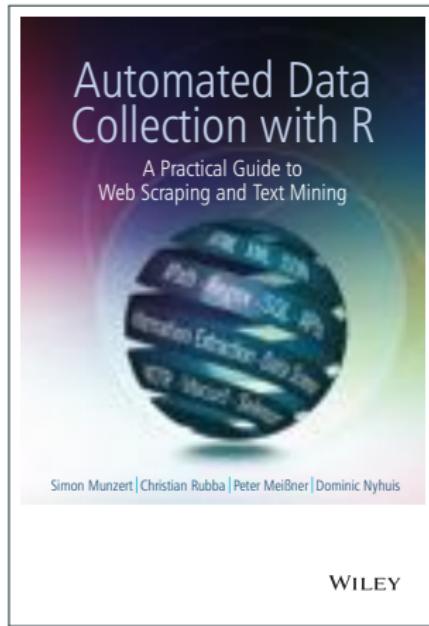
---

<b>Unit</b>	<b>Topic</b>
1	Introduction – setup and first steps
2	Regular expressions
3	Scraping static webpages
4	Advanced scraping of static webpages
5	Scraping dynamic webpages
6	Tapping APIs
7	Legal and ethical issues in web scraping
8	Scraping workflow and tricks of the trade

---

# The accompanying book

- contains most of which I tell you during the course (but much more, and at times more accurate)
- written between 2012 and 2014 → not entirely up-to-date anymore, but I will provide updated material during the course
- homepage with materials: [www.r-datacollection.com](http://www.r-datacollection.com)
- if you find any errors in the book, please let me know!



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Overview

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# **Web scraping with R**

---

# Web scraping. What? Why?

## Web scraping

A.k.a. screen scraping, is the business of

- getting (unstructured) data from the web and
- bringing it into shape (e.g., clean, make tabular format)

## A data analyst's view

- data abundance online
- social interaction online
- services track social behavior

## A pragmatist's view

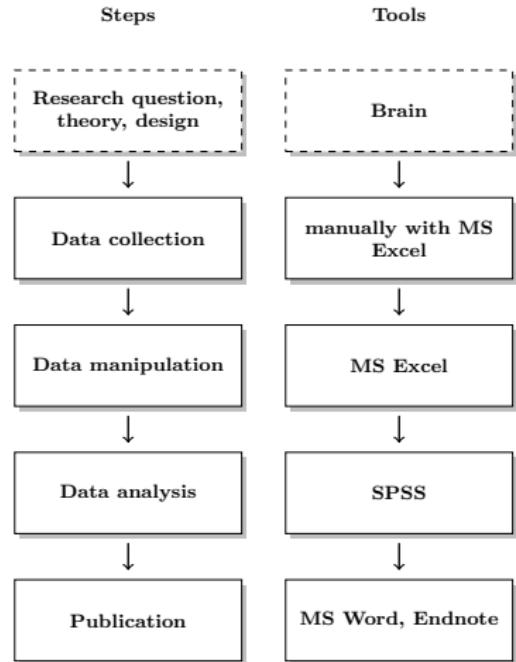
- financial resources
- time resources
- reproducibility
- updateability

# **Why R?**

---

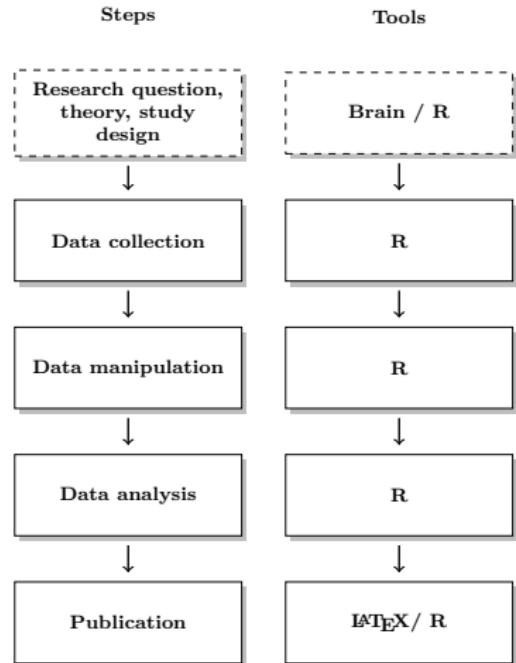
# Why R?

- free
- open source
- large community
- powerful tools for statistical analysis
- powerful tools for visualization
- flexible in processing all kinds of data/languages
- useful in every step of the workflow



# Why R?

- free
- open source
- large community
- powerful tools for statistical analysis
- powerful tools for visualization
- flexible in processing all kinds of data/languages
- useful in every step of the workflow



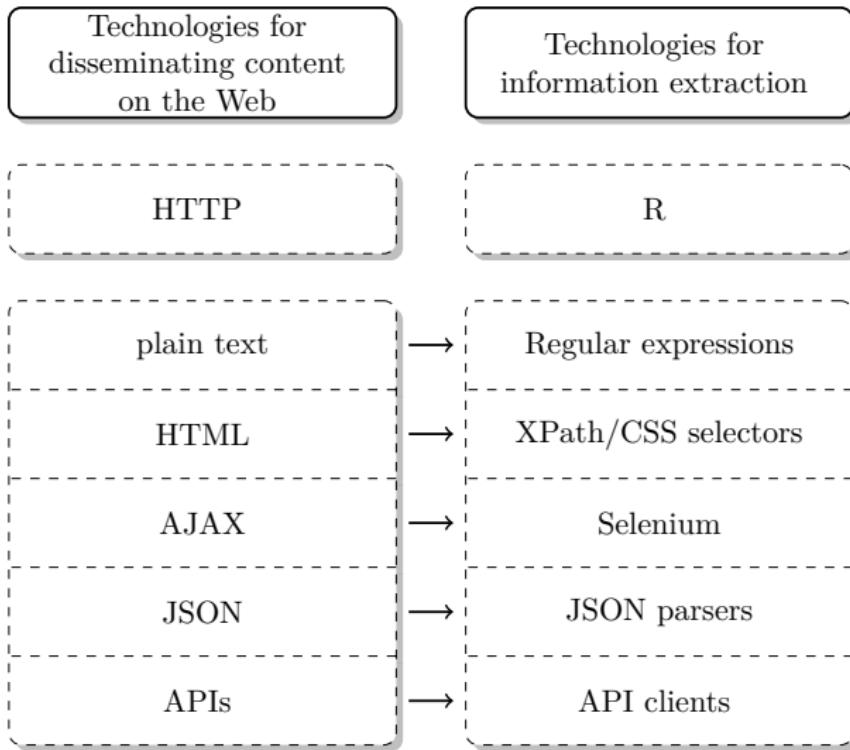
# The philosophy behind web data collection with R

- no point-and-click procedure
- script the entire process from start to finish
- automation of
  - downloading
  - classical screen scraping
  - tapping APIs
  - parsing
  - data tidying, text data processing
- scaling up scraping procedures
- scheduling of scraping tasks

# **Technologies of the World Wide Web**

---

# Technologies of the World Wide Web



# Technical setup

1. make sure that the newest version of R is installed on your computer (available here: <https://cran.r-project.org/>)
2. install the newest stable version of *RStudio Desktop* (available here: <https://www.rstudio.com/products/rstudio/download>)

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

An Introductory Case Study

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Data on the web

The screenshot shows a web browser window with the URL <https://en.wikipedia.org/w/index.php?title=Berlin&oldid=991111199>. The page title is "Berlin". The main content area describes Berlin as the capital and largest city of Germany, with a population of approximately 3.5 million people. It highlights Berlin's status as the second-most populous city proper and the seventh-most populous urban area in the European Union. The page notes its location in north-eastern Germany on the banks of rivers Spree and Havel, and its role as the centre of the Berlin-Brandenburg Metropolitan Region, which has about 6 million residents from more than 180 nations. The text mentions Berlin's temperate oceanic climate and its diverse flora and fauna. A sidebar on the right contains a "Berlin" section with a "State of Germany" heading and several thumbnail images of Berlin landmarks like the Brandenburg Gate and Reichstag.

Berlin

From Wikipedia, the free encyclopedia

This article is about the capital of Germany. For other uses, see [Berlin \(disambiguation\)](#).

**Berlin** (German: [bɛɐ̯lɪn] or [bɛɐ̯lɪn̩]; German Low Saxon: [bɛɐ̯lɪn̩]) is the capital and the largest city of Germany as well as one of its constituent 16 states.<sup>1</sup> With a population of approximately 3.5 million people,<sup>2</sup> Berlin is the second-most populous city proper and the seventh-most populous urban area in the European Union.<sup>3</sup> Located in north-eastern Germany on the banks of rivers Spree and Havel, it is the centre of the Berlin-Brandenburg Metropolitan Region, which has about 6 million residents from more than 180 nations.<sup>4</sup> Due to its location in the European Plain, Berlin is influenced by a temperate oceanic climate. Around one third of the city's area is composed of forests, parks, gardens, rivers and lakes.<sup>5</sup>

First documented in the 13th century and situated at the crossing of two important ancient trade routes,<sup>6</sup> Berlin became the capital of the Margraviate of Brandenburg (1415–1701), the Kingdom of Prussia (1701–1918), the German Empire (1871–1918), the Weimar Republic (1918–1933) and the Third Reich (1933–1945).<sup>7</sup> Berlin in the 1920s was the third largest municipality in the world.<sup>8</sup> After World War II and its consequent occupation by the victorious countries, the city was divided. East Berlin became the capital of East Germany while West Berlin became a de facto West German exclave, surrounded by the Berlin Wall (1961–1989). Following German reunification in

Wikipedia

Contents

Featured content

Recent events

Random article

George to Wikipedia

Wikipedia news

Help

About Wikipedia

Community portal

Recent changes

Commons

Tools

What links here

Recent changes

Upload file

Special pages

Printable version

Page information

Wikidata item

One page

Permanent link

<https://en.wikipedia.org/w/index.php?title=Berlin&oldid=991111199>

## Data on the web

Berlin's 11 sister cities are:

- **Twin towns – sister cities** [edit]
- See also: List of twin towns and sister cities in Germany

Berlin maintains official partnerships with 11 cities.<sup>101</sup> Twin-linking between Berlin and other cities began with its sister city Los Angeles in 1961. East Berlin's partnerships were canceled at the time of German reunification but later partially reestablished. West Berlin's partnerships had previously been restricted to the borough level; during the Cold War era, the partnerships had reflected the different power blocs, with West Berlin partnering with capitals in the Western World, and East Berlin mostly partnering with cities from the Warsaw Pact and its allies.

There are several joint projects with many other cities, such as Berna, Brugge, Brixen, Bratislava, Copenhagen, Florence, Johannesburg, Mumbai, Oslo, Strasbourg, Szeged, Sofia, Sydney, New York City and Vienna. Berlin participates in international city associations such as the Union of the Capitals of the European Union, Eurocities, Network of European Cities of Culture, Metropolis, Summit Conference of the World's Major Cities, and Conference of the World's Capital Cities. Berlin's sister cities are:

• 1947 ■ Los Angeles, United States	• 1952 ■ Brussels, Belgium	• 1964 ■ Tokyo, Japan
• 1947 ■ Paris, France	• 1962 ■ Budapest, Hungary <sup>102</sup>	• 1964 ■ Buenos Aires, Argentina
• 1968 ■ Madrid, Spain	• 1983 ■ Tallinn, Estonia	• 1985 ■ Prague, Czech Republic <sup>103</sup>
• 1989 ■ Istanbul, Turkey	• 1985 ■ Mexico City, Mexico	• 2000 ■ Daegu, South Korea
• 1991 ■ Warsaw, Poland <sup>104</sup>	• 1993 ■ Jakarta, Indonesia	• 2000 ■ London, United Kingdom
• 1991 ■ Moscow, Russia	• 1994 ■ Beijing, China	

**Capital city** [edit]

Berlin is the capital of the Federal Republic of Germany. The President of Germany, whose functions are mainly ceremonial under the German constitution, has his official residence in Schloss Bellevue.<sup>105</sup> Berlin is the seat of the German executive, housed in the Chancellery, the *Bundeskanzleramt*. Facing the Chancellery in the *Bundestag*, the German Parliament, housed in the renovated Reichstag building since the government relocated to Berlin in 1999. The *Bundesrat* ('Federal council'), performing the function of an upper house in the representation of the Federal States (Bundesländer) of Germany and free

# Data on the web

Screenshot of a browser showing the Wikipedia page for Berlin's Twin towns.

The page lists Berlin's official partnerships with 17 cities, starting with Los Angeles in 1967. The partnerships were re-established in 1991 after German reunification. The partnerships are overseen by the city's Governing Mayor and address the issues. The neighbourhoods have no local government bodies.

**Twin towns – Sister cities** [edit]

(See also List of twin towns and sister cities in Germany)

Berlin maintains official partnerships with 17 cities.<sup>1007</sup> Twin-linking between Berlin and other cities began with its sister-city Los Angeles in 1967. East Berlin's partnerships were re-established at the time of German reunification but were partially reinstated. West Berlin's partnerships had previously been restricted to the borough level. During the Cold War era, the partnerships had reflected the different power blocs, with West Berlin partnering with capitals in the Western World, and East Berlin mostly partnering with cities from the Warsaw Pact and its allies.

There are several joint projects with many other cities, such as Bern, Brugge, São Paulo, Copenhagen, Hiroshima, Johannesburg, Mumbai, Oslo, Birmingham, Seoul, Sofia, Sydney, New York City and Vienna. Berlin participates in international city associations such as the Union of the Capitals of the European Union, Eurocities, Network of European Cities of Culture, Metropolis, Summit Conference of the World's Major Cities, and Conference of the World's Capital Cities. Berlin's offices were office city<sup>1008</sup>

→ 1967  <b>Los Angeles, United States</b>	→ 1980  <b>Brussels, Belgium</b>	→ 1998  <b>Tokyo, Japan</b>
→ 1981  <b>Paris, France</b>	→ 1982  <b>Budapest, Hungary</b> <sup>1009</sup>	→ 1999  <b>Buenos Aires, Argentina</b>
→ 1989  <b>Madrid, Spain</b>	→ 1993  <b>Vilnius, Lithuania</b>	→ 1999  <b>Prague, Czech Republic</b> <sup>1010</sup>
→ 1991  <b>Istanbul, Turkey</b>	→ 1993  <b>Mexico City, Mexico</b>	→ 2000  <b>Windhoek, Namibia</b>
→ 1991  <b>Lisbon, Portugal</b> <sup>1011</sup>	→ 1994  <b>Jakarta, Indonesia</b>	→ 2000  <b>London, United Kingdom</b>
→ 1991  <b>Moscow, Russia</b>	→ 1996  <b>Beijing, China</b>	

**Capital city** [edit]

Berlin is the capital of the Federal Republic of Germany. The President of Germany, whose functions are mainly ceremonial under the German constitution, has his official residence in Schloss Bellevue.<sup>1012</sup> Berlin is the seat of the German executive. Housed in the Chancellery, the Bundeskanzleramt, facing the Chancellery in the Bundestag, the German Parliament, housed in the renovated Reichstag building since the government relocated to Berlin in 1990. The Bundesrat ("Federal council"), performing the function of an upper house in the representation of the Federal States (Bundesländer) in Germany and thus its

# Let's grab these data!

## Step 1: Load packages

R code —

```
1 library(rvest)  
2 library(stringr)
```

— end

# Let's grab these data!

## Step 2: Parse page source

R code —

```
3 library(rvest)
4 library(stringr)
5 parsed_url <- read_html("https://en.wikipedia.org/wiki/Berlin")
```

— end

# Let's grab these data!

---

## Step 3: Extract information

R code —

---

```
6 library(rvest)
7 library(stringr)
8 parsed_url <- read_html("https://en.wikipedia.org/wiki/Berlin")
9 parsed_nodes <- html_nodes(parsed_url, xpath = "//div[contains(@class,
  'column-count-3')]/li")
10 cities <- html_text(parsed_nodes)
11 cities[1:10]
[1] "1967 Los Angeles, United States" "1987 Paris, France"
[3] "1988 Madrid, Spain"                  "1989 Istanbul, Turkey"
[5] "1991 Warsaw, Poland[103]"           "1991 Moscow, Russia"
[7] "1992 Brussels, Belgium"              "1992 Budapest, Hungary[104]"
[9] "1993 Tashkent, Uzbekistan"          "1993 Mexico City, Mexico"
```

---

end

# Why was this so easy?

The screenshot shows a web browser displaying the Wikipedia article "List of the Capitals of the European Union". The page contains a table of capital cities for various countries, grouped by their status in the European Union (EU, EEA, or EFTA). The browser's developer tools are open, with the DOM tree panel selected. A tooltip from the browser's context menu is displayed over the entry for Paris.

Category	Capital City	Country
EU	Brussels	Belgium
	Vienna	Austria
	Bratislava	Slovakia
	Sofia	Bulgaria
	Madrid	Spain
	Paris	France
	Brussels	Belgium
	Helsinki	Finland
	Vienna	Austria
	Bratislava	Slovakia
	Sofia	Bulgaria
	Madrid	Spain
EEA	Tallinn	Estonia
	Riga	Lithuania
	Oslo	Norway
	Edinburgh	United Kingdom
	Brussels	Belgium
	Vienna	Austria
	Bratislava	Slovakia
	Sofia	Bulgaria
	Madrid	Spain
	Paris	France
EFTA	London	United Kingdom
	Edinburgh	United Kingdom
	Brussels	Belgium
	Vienna	Austria
	Bratislava	Slovakia
	Sofia	Bulgaria
	Madrid	Spain
	Paris	France
	London	United Kingdom
	Edinburgh	United Kingdom

# Why was this so easy?



# Why was this so easy?

The screenshot shows a web browser window with two tabs open. The left tab displays a list of European capitals with their flags and names. A red arrow points to the right-click context menu on one of the entries, with the text "right-click, „inspect element...“" overlaid. The right tab shows the HTML source code for the same page, with another red arrow pointing to the "View HTML source code" button at the top of the developer tools interface.

right-click,  
„inspect  
element...“

view HTML source code

list of European capitals:

- 1990: Berlin, West Berlin, East Berlin, Bonn
- 1991: Paris, France
- 1992: Madrid, Spain
- 1993: Warsaw, Poland
- 1994: Ankara, Turkey
- 1995: Brussels, Belgium
- 1996: Budapest, Hungary
- 1997: Bern, Switzerland
- 1998: Rome, Italy
- 1999: Prague, Czech Republic
- 2000: Madrid, Spain
- 2001: Lisbon, Portugal
- 2002: London, United Kingdom

Capital city (←)

Berlin is the capital of the Federal Republic of Germany. The Presidents of Germany, whose functions are mainly ceremonial under the German constitution,

HTML source code:

```
<ul style="list-style-type: none; padding-left: 0; margin: 0; border: 1px solid black; border-radius: 5px; background-color: #f9f9f9; overflow: auto; width: 100%;">
    <li>1990: Berlin, West Berlin, East Berlin, Bonn</li>
    <li>1991: Paris, France</li>
    <li>1992: Madrid, Spain</li>
    <li>1993: Warsaw, Poland</li>
    <li>1994: Ankara, Turkey</li>
    <li>1995: Brussels, Belgium</li>
    <li>1996: Budapest, Hungary</li>
    <li>1997: Bern, Switzerland</li>
    <li>1998: Rome, Italy</li>
    <li>1999: Prague, Czech Republic</li>
    <li>2000: Madrid, Spain</li>
    <li>2001: Lisbon, Portugal</li>
    <li>2002: London, United Kingdom</li>


```

# Why was this so easy?

The screenshot shows a web browser window with two tabs open. The left tab displays a list of European capitals with their flags and names. The right tab shows the HTML source code for the same page. A red box highlights the 'view HTML source code' button at the top of the browser. Another red box highlights the 'right-click „Inspect element...“' instruction, which is pointing to the context menu of a specific element in the left list. A third red box highlights the 'Identify elements that store information of interest' instruction, pointing to the 'id="list"' attribute in the source code.

right-click „Inspect element...“

view HTML source code

Identify elements that store information of interest

```
<ul id="list">
```

# Let's clean up these data!

R code

---

```
12 cities[1:10]
[1] "1967 Los Angeles, United States" "1987 Paris, France"
[3] "1988 Madrid, Spain"                 "1989 Istanbul, Turkey"
[5] "1991 Warsaw, Poland[103]"          "1991 Moscow, Russia"
[7] "1992 Brussels, Belgium"            "1992 Budapest, Hungary[104]"
[9] "1993 Tashkent, Uzbekistan"         "1993 Mexico City, Mexico"
```

---

end

## Step 1: Remove footnotes with a regular expression

R code

---

```
13 cities <- str_replace(cities, "\\[\\d+\\]", "")
14 cities[1:10]
[1] "1967 Los Angeles, United States" "1987 Paris, France"
[3] "1988 Madrid, Spain"                 "1989 Istanbul, Turkey"
[5] "1991 Warsaw, Poland"               "1991 Moscow, Russia"
[7] "1992 Brussels, Belgium"            "1992 Budapest, Hungary"
[9] "1993 Tashkent, Uzbekistan"         "1993 Mexico City, Mexico"
```

---

end

# Let's clean up these data!

## Step 2: Extract data with regular expressions

R code —

---

```
15 year <- str_extract(cities, "\\d{4}")  
16 city <- str_extract(cities, "[[:alpha:] ]+") %>% str_trim  
17 country <- str_extract(cities, "[[:alpha:] ]+$") %>% str_trim  
18 year[1:10]  
[1] "1967" "1987" "1988" "1989" "1991" "1991" "1992" "1992" "1993" "  
1993"  
19 city[1:10]  
[1] "Los Angeles" "Paris"          "Madrid"        "Istanbul"      "Warsaw"  
[6] "Moscow"       "Brussels"       "Budapest"     "Tashkent"      "Mexico  
City"  
20 country[1:10]  
[1] "United States" "France"       "Spain"        "Turkey"  
[5] "Poland"        "Russia"       "Belgium"     "Hungary"  
[9] "Uzbekistan"    "Mexico"
```

end

# Let's clean up these data!

---

## Step 3: Put everything into data frame

R code —

---

```
21 cities_df <- data.frame(year, city, country)  
22 head(cities_df)
```

	year	city	country
1	1967	Los Angeles	United States
2	1987	Paris	France
3	1988	Madrid	Spain
4	1989	Istanbul	Turkey
5	1991	Warsaw	Poland
6	1991	Moscow	Russia

---

end

# Let's map these data!

---

## Step 1: Load necessary packages

R code —

---

```
23 library(ggmap)  
24 library(maps)
```

---

end

# Let's map these data!

## Step 2: Geocode cities with the Google Maps API

R code —

---

```
25 library(ggmap)
26 library(maps)
27 cities_coords <- geocode(paste0(cities_df$city, ", ", cities_df$country)
28 )  
29 cities_df$lon <- cities_coords$lon
30 cities_df$lat <- cities_coords$lat
31 cities_df$lon[1:10]
 [1] -118.243685  2.352222 -3.703790 28.978359      NA
 [6]   37.617300      NA   19.040235      NA -99.133208
31 cities_df$lat[1:10]
 [1] 34.05223 48.85661 40.41678 41.00824      NA 55.75583      NA
 [8] 47.49791      NA 19.43261
```

---

end

# Let's map these data!

## Step 3: Plot world map, add coordinates

R code

```
32 map_world <- borders("world", colour = "gray50", fill = "white")
33 ggplot() + map_world + geom_point(aes(x = cities_df$lon, y = cities_df$lat), color = "red", size = 1) + theme_void()
```

end



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

HTML

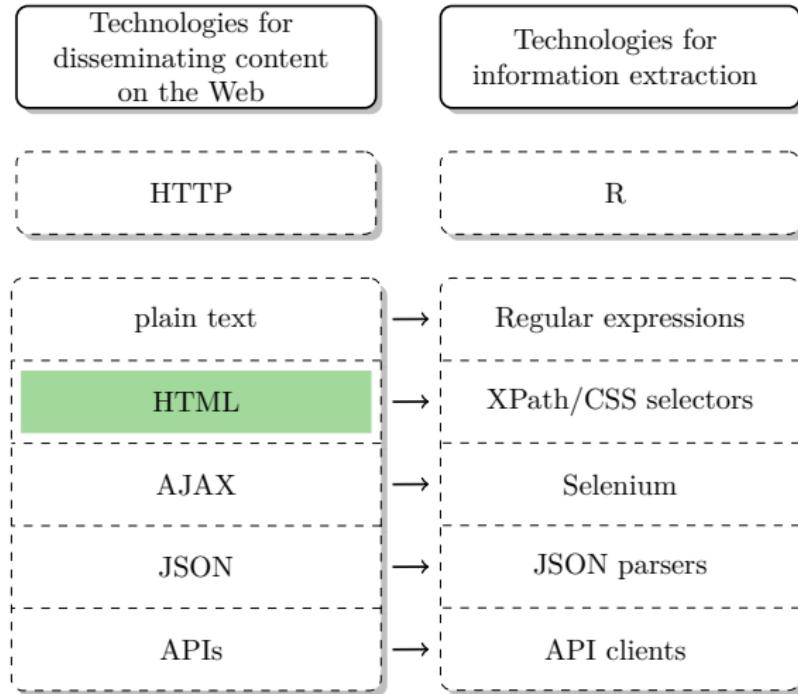
---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Technologies of the World Wide Web



# **HTML basics**

---

# HTML – a quick primer

## What's HTML?

- HyperText Markup Language
- markup language = plain text + markups
- W3C standard for the construction of websites
- lies underneath of what you see in your browser

## Why is this important to us?

- it determines where and how information is stored
- a basic understanding of HTML helps us locate the information we want to retrieve
- relax. A passive understanding of HTML is sufficient

# HTML in the wild

The screenshot shows a standard web browser interface with the URL <https://en.wikipedia.org/w/index.php?title=Berlin&oldid=5811109>. The main content is the article about Berlin, Germany. The page includes a sidebar with navigation links like 'Recent changes' and 'Community portal'. On the right, there's a sidebar titled 'Berlin' showing a thumbnail of the Brandenburg Gate and other Berlin-related images.

## Berlin

From Wikipedia, the free encyclopedia

This article is about the capital of Germany. For other uses, see Berlin (disambiguation).

**Berlin** (German: [ˈbɛʁlɪn] (listen)) is the capital and the largest city of Germany as well as one of its constituent 16 states. With a population of approximately 3.6 million people,<sup>20</sup> Berlin is the second-most populous city proper and the seventh-most populous urban area in the European Union.<sup>21</sup> Located in northeastern Germany on the banks of rivers Spree and Havel, it is the centre of the Berlin/Brandenburg Metropolitan Region, which has about 8 million residents from more than 180 nations.<sup>22</sup> Due to its location in the European Plain, Berlin is influenced by a temperate seasonal climate. Around one third of the city area is composed of forests, parks, gardens, rivers and canals.<sup>23</sup>

First documented in the 13th century and situated at the crossing of two important future trade routes,<sup>24</sup> Berlin became the capital of the Margravate of Brandenburg (1415–1701), the Kingdom of Prussia (1701–1918), the German Empire (1871–1918), the Weimar Republic (1919–1933) and the Third Reich (1933–1945).<sup>25</sup> Berlin in the 1920s was the third largest municipality in the world.<sup>26</sup> After World War II and its consequent occupation by the victorious countries, the city was divided; East Berlin became the capital of East Germany while West Berlin became a de facto West German exclave, surrounded by the Berlin Wall (1961–1989). Following Germany's reunification in 1990, Berlin once again became the capital of统一的Germany.

# HTML in the wild

Sicher | https://en.wikipedia.org/w/index.php?title=Berlin&oldid=970000000

which is led by the city's Governing Mayor and advises the Senate. The neighbourhoods have no local government bodies.

### Twin towns -- sister cities [edit]

(See also: List of twin towns and sister cities in Germany)

Berlin maintains official partnerships with 11 cities.<sup>[107]</sup> Twin-linking between Berlin and other cities began with its sister city Los Angeles in 1961. East Berlin's partnerships were dissolved at the time of German reunification but later partially reinstated. West Berlin's partnerships had previously been restricted to the borough level. During the Cold War era, the partnerships had reflected the different power-blocks, with West Berlin partnering with capitals in the Western World, and East Berlin mostly partnering with cities from the Warsaw Pact and its allies.

There are several joint projects with many other cities, such as Belgrade, Brasília, São Paulo, Copenhagen, Hanover, Johannesburg, Manila, Oslo, Shanghai, Seoul, Sofia, Sydney, New York City and Vienna. Berlin participates in international city associations such as the Union of the Capitals of the European Union, Eurocities, Network of European Cities of Culture, Metropolis, Summit Conference of the World's Major Cities, and Conference of the World's Capital Cities. Berlin's official sister cities are:<sup>[108]</sup>

– 1967 ■■■ Los Angeles, United States	– 1962 ■■■ Brussels, Belgium	– 1964 ■ Tokyo, Japan
– 1967 ■■■ Paris, France	– 1962 ■■■ Budapest, Hungary <sup>[109]</sup>	– 1964 ■■■ Buenos Aires, Argentina
– 1968 ■■■ Madrid, Spain	– 1965 ■■■ Tel Aviv, Israel	– 1969 ■■■ Prague, Czech Republic <sup>[110]</sup>
– 1969 ■■■ Istanbul, Turkey	– 1966 ■■■ Mexico City, Mexico	– 2000 ■■■ Windhoek, Namibia
– 1981 ■■■ Warsaw, Poland <sup>[111]</sup>	– 1966 ■■■ Jakarta, Indonesia	– 2001 ■■■ London, United Kingdom
– 1987 ■■■ Moscow, Russia	– 1994 ■■■ Beijing, China	

### Capital city [edit]

Berlin is the capital of the Federal Republic of Germany. The President of Germany, whose functions are mainly ceremonial under the German constitution, has his official residence in Schloss Bellevue.<sup>[112]</sup> Berlin is the seat of the German executive, housed in the Chancellery, the Bundeskanzleramt. Facing the Chancellery is the Bundestag, the German Parliament, housed in the renovated Reichstag building since the government relocated to Berlin in 1990. The Bundesrat ('federal council'), performing the function of an upper house in the representation of the Federal States (Bundesländer) of Germany and has its

# HTML in the wild

The screenshot shows a web browser window with the URL [https://en.wikipedia.org/w/index.php?title=List\\_of\\_capitals\\_of\\_the\\_European\\_Union&oldid=10500000](https://en.wikipedia.org/w/index.php?title=List_of_capitals_of_the_European_Union&oldid=10500000). The main content area displays a list of years from 1984 to 2008, each associated with a country and its capital city. The right side of the screen shows the browser's developer tools, specifically the Elements tab, which highlights the HTML structure of the page.

The list of years and their corresponding countries and capitals is as follows:

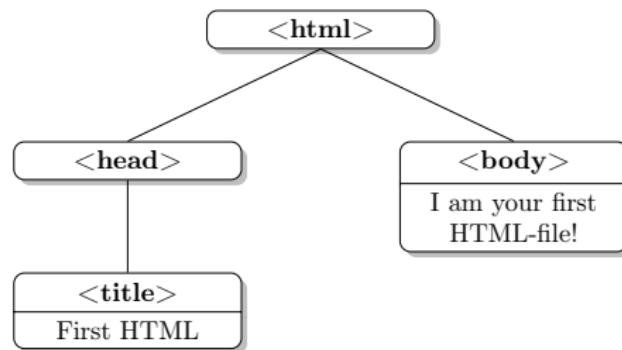
- 1984: Brussels, Belgium
- 1985: Tokyo, Japan
- 1986: Lisbon, Portugal
- 1987: Warsaw, Poland
- 1988: Athens, Greece
- 1989: Bucharest, Romania
- 1990: Sarajevo, Bosnia and Herzegovina
- 1991: Madrid, Spain
- 1992: Prague, Czech Republic
- 1993: Copenhagen, Denmark
- 1994: Vilnius, Lithuania
- 1995: Stockholm, Sweden
- 1996: Reykjavík, Iceland
- 1997: Warsaw (2), Poland
- 1998: Belgrade, Serbia
- 1999: Ljubljana, Slovenia
- 2000: Madrid (2), Spain
- 2001: Ankara, Turkey
- 2002: Bucharest (2), Romania
- 2003: Berlin, Germany
- 2004: Athens (2), Greece
- 2005: Prague (2), Czech Republic
- 2006: Ljubljana (2), Slovenia
- 2007: Bucharest (3), Romania
- 2008: Beijing, China
- 2009: London, United Kingdom

**Capital city** (edit)

Berlin is the capital of the Federal Republic of Germany. The President of Germany, whose functions are mainly ceremonial under the German constitution,

# Tree structure

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title id=1>First HTML</title>
5   </head>
6   <body>
7     I am your first HTML file!
8   </body>
9 </html>
```



## **Elements and attributes**

---

# Elements and attributes

## Elements

Elements are a combination of *start tags*, content, and *end tags*.

Example:

```
1 <title>First HTML</title>
```

## Syntax

element title	title
start tag	<title>
end tag	</title>
value	First HTML

# Elements and attributes

## Attributes

Attributes describe elements and are stored in the start tag. In HTML, there are specific attributes for specific elements.

Example:

```
1 <a href="http://www.r-datacollection.com/">Link to Homepage</a>
```

## Syntax

- name-value pairs: `name="value"`
- simple and double quotation marks possible
- several attributes per element possible

## Why tags and attributes are important

- tags structure HTML documents
- everything that structures a document can be used to extract information
- in the following, we get to know some important tags which are useful when scraping information from the Web

# Important tags and attributes

## Anchor tag <a>

- links to other pages or resources
- classical links are always formatted with an anchor tag
- the **href** attribute determines the target location
- the value is the name of the link

Link to another resource:

```
1 <a href="en.wikipedia.org/wiki/List_of_lists_of_lists">Link with absolute path</a>
```

Reference in a document:

```
1 <a id="top">Reference Point</a>
```

Link to a reference:

```
1 <a href="#top">Link to Reference Point</a>
```

# Important tags and attributes

## Heading tags `<h1>`, `<h2>`, ..., and paragraph tag `<p>`

- structure text and paragraphs
- heading tags range from level 1 to 6
- paragraph tag induces line break

Examples:

```
1 <p>This text is going to be a paragraph one day and separated from other  
2 text by line breaks.</p>
```

```
1 <h1>heading of level 1 -- this will be BIG</h1>  
2 ...  
3 <h6>heading of level 6 -- the smallest heading</h6>
```

# Important tags and attributes

## **Listing tags <ul>, <ol> and <dl>**

- the `<ol>` tag creates a numeric list, `<ul>` an unnumbered list, `<dl>` a definition list
- list elements are indicated with the `<li>` tag

Example:

```
1 <ul>
2   ^^I<li>Dogs</li>
3   ^^I<li>Cats</li>
4   ^^I<li>Fish</li>
5 </ul>
```

# Important tags and attributes

## Organizational tags `<div>` and `<span>`

- grouping of content over lines (`<div>`) or within lines (`<span>`)
- do not change the layout themselves but work together with CSS

### Example of CSS definition

```
1  div.happy { color:pink;  
2      font-family:"Comic Sans MS";  
3      font-size:120% }  
4  span.happy { color:pink;  
5      font-family:"Comic Sans MS";  
6      font-size:120% }
```

### In the HTML document

```
1  <div class="happy"><p>I am a happy styled paragraph</p></div>  
2  non-happy text with <span class="happy">some happiness</span>
```

# Important tags and attributes

## Form tag <form>

- allows to incorporate HTML forms
- client can send information to the HTTP server via forms
- whenever you type something into a field or click on radio buttons in your browser, you are interacting with forms

Example:

```
1 <form name="submitPW" action="Passed.html" method="get">
2   password:
3   <input name="pw" type="text" value="">
4   <input type="submit" value="SubmitButtonText">
5 </form>
```

# Important tags and attributes

## Table tags `<table>`, `<tr>`, `<td>`, and `<th>`

- standard HTML tables always follow a standard architecture
- the different tags allow to define the table as a whole, individual rows (including the heading), and cells
- if the data is hidden in tables, scraping will be straightforward

Example:

```
1 <table>
2   <tr> <th>Rank</th> <th>Nominal GDP</th> <th>Name</th> </tr>
3   <tr> <th></th> <th>(per capita, USD)</th> <th></th> </tr>
4   <tr> <td>1</td> <td>170,373</td> <td>Lichtenstein</td> </tr>
5   <tr> <td>2</td> <td>167,021</td> <td>Monaco</td> </tr>
6   <tr> <td>3</td> <td>115,377</td> <td>Luxembourg</td> </tr>
7   <tr> <td>4</td> <td>98,565</td> <td>Norway</td> </tr>
8   <tr> <td>5</td> <td>92,682</td> <td>Qatar</td> </tr>
9 </table>
```

## **Summary**

---

# Summary

- HTML is the *lingua franca* on the web
- content on webpages is structured by HTML tags that are nested in a tree structure
- to break open information, we will have to locate it in the HTML tree
- for web scraping purposes, a mostly passive knowledge of HTML is sufficient

```
<DIV>Q: HOW DO YOU ANNOY A WEB DEVELOPER?</SPAN>
```

Source: <https://xkcd.com/1144/> (Randall Munroe)

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Regular Expressions Basics

---

Simon Munzert

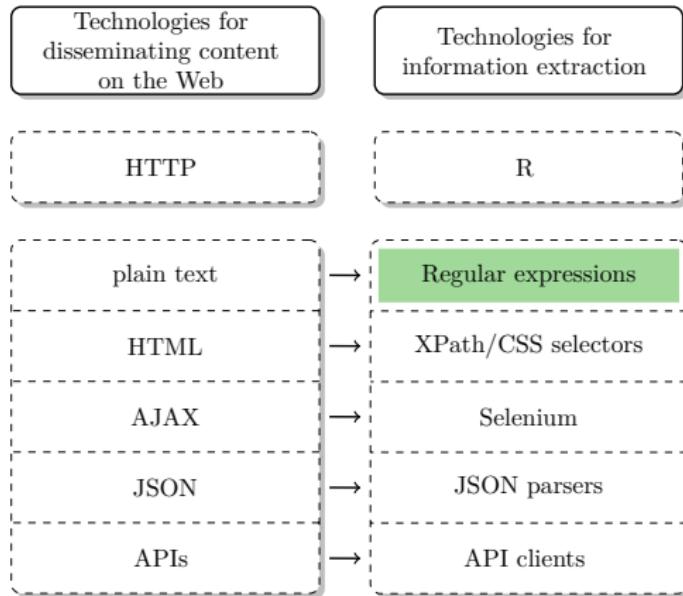
Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# **Regular expressions**

---

# Technologies of the World Wide Web



# What are regular expressions?

## Definition

- a.k.a. *regex* or *RegExp*
- origins in formal language theory
- sequences of characters that describe patterns in text
- implemented in many programming languages, including R

## Why are regular expressions useful for web scraping?

- information on the web can often be described by patterns (think email addresses, numbers, cells in HTML tables, ...)
- if the data of interest follow specific patterns, we can match and extract them—regardless of page layout and HTML overhead
- whenever the information of interest is (stored in) text, regular expressions are useful for extraction and tidying purposes

## **Introductory example**

---

# Introductory example

R code

---

```
1 raw.data <- "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
2 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
3 Simpson,Homer5553642Dr. Julius Hibbert"
```

---

end

- vector `raw.data` contains unstructured phonebook entries
- goal: extraction of entries
- problem: find a pattern that matches names and numbers
- solution: regex!

# Introductory example

R code

---

```
4 raw.data <- "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
5 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
6 Simpson,Homer5553642Dr. Julius Hibbert"
```

---

end

Solution:

- load package **stringr** (more on that later)
- a detective's work: construct regex for names
- apply regex on raw vector

R code

---

```
7 library(stringr)  
8 name <- unlist(str_extract_all(raw.data, "[[:alpha:].[[:alpha:]]{2,}"))  
9 name  
[1] "Moe Szyslak"           "Burns, C. Montgomery" "Rev. Timothy Lovejoy"  
[4] "Ned Flanders"          "Simpson,Homer"        "Dr. Julius Hibbert"
```

---

end

# Introductory example

Solution, continued:

- construct and apply regex for phone numbers
- combine both vectors

R code

---

```
10 phone <- unlist(str_extract_all(raw.data, "\\\(?(\\\d{3})?\\)?)?(-| )?\\d{3}(-| )?\\d{4}"))
11 phone
[1] "555-1239"          "(636) 555-0113" "555-6542"           "555 8904"
[5] "636-555-3226"      "5553642"
12 data.frame(name = name, phone = phone)
            name        phone
1       Moe Szyslak    555-1239
2 Burns, C. Montgomery (636) 555-0113
3 Rev. Timothy Lovejoy    555-6542
4       Ned Flanders    555 8904
5       Simpson,Homer   636-555-3226
6     Dr. Julius Hibbert 5553642
```

end

## Summary



Source: <https://xkcd.com/208/> (Randall Munroe)

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Regular Expressions in R

---

Simon Munzert

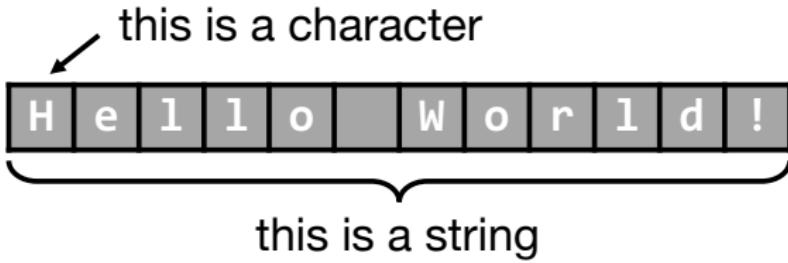
Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# **Regular expressions in R**

---

# Regular expressions in R



# Regular expressions in R

An example string:

R code

---

```
1 example.obj <- "1. A small sentence. - 2. Another tiny sentence."  
-----  
end
```

We are going to use the `str_extract()` function and the `str_extract_all()` function from the `stringr` package to apply regular expressions in strings. The generic syntax is:

```
str_extract(string, pattern)  
str_extract_all(string, pattern)
```

`str_extract()` returns the first match, `str_extract_all()` returns all matches.

## Strings match themselves

R code

```
2 str_extract(example.obj, "small")
[1] "small"
3 str_extract(example.obj, "banana")
[1] NA
```

end

## Multiple matches are returned as a list

R code

```
4 (out <- str_extract_all(c("text", "manipulation", "basics"), "a"))
[[1]]
character(0)

[[2]]
[1] "a" "a"

[[3]]
[1] "a"
```

end

# Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

## Character matching is case sensitive

R code

```
5 str_extract(example.obj, "small")
[1] "small"
6 str_extract(example.obj, "SMALL")
[1] NA
7 str_extract(example.obj, regex("SMALL", ignore_case = TRUE))
[1] "small"
```

end

## We can match arbitrary combinations of characters

R code

```
8 str_extract(example.obj, "mall sent")
[1] "mall sent"
```

end

# Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

## Matching the beginning of a string

R code —

```
9 str_extract(example.obj, "^1")
[1] "1"
10 str_extract(example.obj, "^2")
[1] NA
```

— end

## Matching the ending of a string

R code —

```
11 str_extract(example.obj, "sentence$")
[1] NA
12 str_extract(example.obj, "sentence.$")
[1] "sentence."
```

— end

# Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

## Express an "or" with the pipe operator

R code —————

```
13 unlist(str_extract_all(example.obj, "tiny|sentence"))
[1] "sentence" "tiny"      "sentence"
```

————— end

## The dot: the ultimate wildcard

R code —————

```
14 str_extract(example.obj, "sm.11")
[1] "small"
```

————— end

## Matching of meta characters

- some symbols have a special meaning in the regex syntax: `.`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `^`, `$`, `*`, `+`, `?` and `-`.
- if we want to match them literally, we have to use an escape sequence: `\symbol`
- as `\` is a meta character itself, we have to escape it with `\`, so we always write `\\symbol` (weird, isn't it?!)
- alternatively, use `fixed("symbols")` to let the parser interpret a chain of symbols literally

R code

```
15 unlist(str_extract_all(example.obj, "\\\."))  
[1] "." "." "." ".."  
16 unlist(str_extract_all(example.obj, fixed(".")))  
[1] "." "." "." ".."
```

# **Character classes**

---

## Square brackets define character classes

Character classes help define special wild cards. The idea is that any of the characters within the brackets can be matched.

R code

```
17 str_extract(example.obj, "sm[abc]ll")  
[1] "small"
```

end

## The hyphen defines a range of characters

R code

```
18 str_extract(example.obj, "sm[a-p]ll")  
[1] "small"
```

end

## Predefined character classes

<code>[:digit:]</code>	Digits: 0 1 2 3 4 5 6 7 8 9
<code>[:lower:]</code>	Lower-case characters: a–z
<code>[:upper:]</code>	Upper-case characters: A–Z
<code>[:alpha:]</code>	Alphabetic characters: a–z and A–Z
<code>[:alnum:]</code>	Digits and alphabetic characters
<code>[:punct:]</code>	Punctuation characters: ‘,’ ‘;’ etc.
<code>[:graph:]</code>	Graphical characters: <code>[:alnum:]</code> and <code>[:punct:]</code>
<code>[:blank:]</code>	Blank characters: Space and tab
<code>[:space:]</code>	Space characters: Space, tab, newline, etc.
<code>[:print:]</code>	Printable characters: <code>[:alnum:]</code> , <code>[:punct:]</code> , <code>[:space:]</code>

## Predefined character classes in action

R code

```
19 unlist(str_extract_all(example.obj, "[[:punct:]])")
[1] "." "." "-" "." "."
20 unlist(str_extract_all(example.obj, "[[:alpha:]])")
[1] "A" "s" "m" "a" "l" "l" "s" "e" "n" "t" "e" "n" "c" "e" "A" "n" "o"
[18] "t" "h" "e" "r" "t" "i" "n" "y" "s" "e" "n" "t" "e" "n" "c" "e"
```

end

## Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

**Predefined character classes are useful because they are efficient**

- combine different kinds of characters
  - facilitate reading of an expression
  - include special characters, e.g., ß, ö, ...
  - can be extended

R code

```
21 unlist(str_extract_all(example.obj, "[[:punct:]ABC]"))
[1] ":" "A" ":" "-" ":" "A" ":" 
22 unlist(str_extract_all(example.obj, "[^[:alnum:]]"))
[1] ":" " " " " " " " " " " " " " " " " " " " " " " " " " " " "
```

end

## Alternative character classes

---

\w	Word characters: <code>[:alnum:]_]</code>
\W	No word characters: <code>^[:alnum:]_]</code>
\s	Space characters: <code>[:blank:]</code>
\S	No space characters: <code>^[:blank:]</code>
\d	Digits: <code>[:digit:]</code>
\D	No digits: <code>^[:digit:]</code>
\b	Word edge
\B	No word edge
\<	Word beginning
\>	Word end

---

# Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

## Alternative character classes in action

R code

```
23 unlist(str_extract_all(example.obj, "\\w+"))
[1] "1"          "A"          "small"       "sentence"    "2"          "Another"
[7] "tiny"        "sentence"

24 unlist(str_extract_all(example.obj, "e\\b"))
[1] "e" "e"
```

end

## Use of quantifiers

?	The preceding item is optional and will be matched at most once
*	The preceding item will be matched zero or more times
+	The preceding item will be matched one or more times
{n}	The preceding item is matched exactly n times
{n,}	The preceding item is matched n or more times
{n,m}	The preceding item is matched between n and m times

R code

```
25 str_extract(example.obj, "s[[[:alpha:]][[:alpha:]][[:alpha:]]1")  
[1] "small"  
26 str_extract(example.obj, "s[[[:alpha:]]{3}1")  
[1] "small"  
27 str_extract(example.obj, "A.+sentence")  
[1] "A small sentence. - 2. Another tiny sentence"
```

end

## Greedy quantification

- the use of `'.+'` results in 'greedy' matching, i.e. the parser tries to match as many characters as possible
- not always desired, but `'.+?'` helps avoid greedy quantification

R code

---

```
28 str_extract(example.obj, "A.+sentence")
[1] "A small sentence. - 2. Another tiny sentence"
29 str_extract(example.obj, "A.+?sentence")
[1] "A small sentence"
```

---

end

## Meta symbols in character classes

- within a *character class*, most meta symbols lose their special meaning
- exceptions: `^` and `-`
- `-` at the beginning or end matches the hyphen

R code

```
30 unlist(str_extract_all(example.obj, "[1-2]"))
[1] "1" "2"
31 unlist(str_extract_all(example.obj, "[12-]"))
[1] "1" "--" "2"
```

end

# **Backreferencing**

---

## Positive and negative lookahead and lookbehind assertions

- match if a certain pattern before (or after) the actual match is found (or not), but are not returned themselves
- positive and negative look**ahead** assertions:  
`(?=...)` and `(?!...)`
- positive and negative look**behind** assertions:  
`(?<=...)` and `(?<!...)`

R code

```
32 unlist(str_extract_all(example.obj, "(?<=2. ).+"))
[1] "Another tiny sentence."
33 unlist(str_extract_all(example.obj, ".+(?=2)"))
[1] "1. A small sentence. - "
```

end

## Backreferencing

- regular expression ‘with memory’
- repeated match of previously matched pattern
- we refer to the first match (defined with round brackets) using \1, to the second match with \2 etc. (up to 9)

R code

```
34 str_extract(example.obj, "([[:alpha:]]) .+?\\1")  
[1] "A small sentence. - 2. A"
```

end

**Logic:** Match the first letter, then anything until you find the first letter again (not greedy)

# Regular expressions in R

"1. A small sentence. - 2. Another tiny sentence."

## Backreferencing: a bit more complicated

**Goal:** match a word that does not include 'a' until the word appears the second time

### Solution:

R code

```
35 str_extract(example.obj, "(\\b[b-z]+\\b).+?\\1")  
[1] "sentence. - 2. Another tiny sentence"
```

end

### Mechanic:

1. match all letters without 'a': [b-z]+
2. match complete words (have beginning and end): \\b
3. refer to the word: ()
4. match anything in between: .+?\\1

## **Summary**

---

# Summary

- regular expressions are magic
- regular expressions are non-trivial
- regular expressions are unreadable
- the good news: for scraping purposes, we can (and should!) rely on other, simpler and more robust tools
- still, regex can prove incredibly powerful under certain circumstances

**CODING HORROR**



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

## String Manipulation

---

Simon Munzert

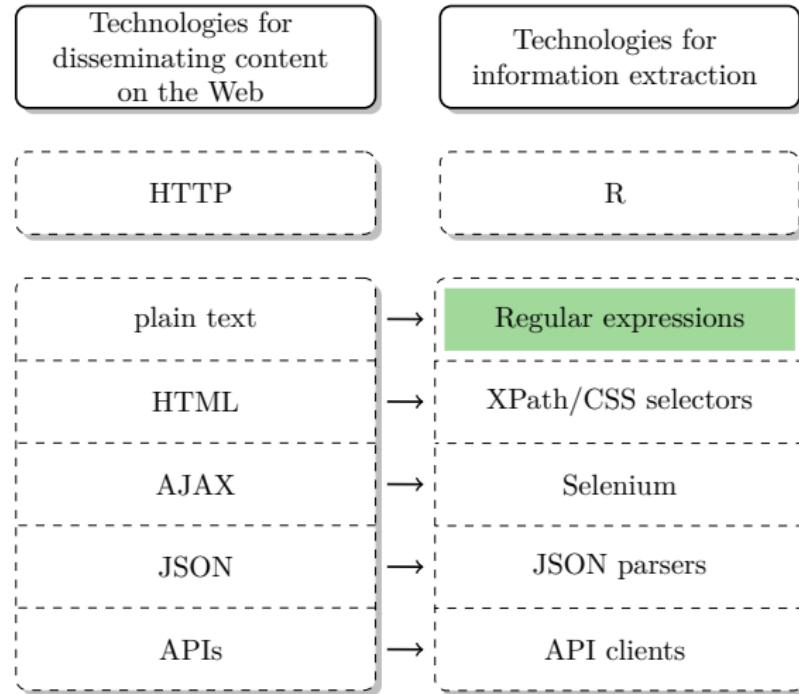
Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **String manipulation in R**

---

# Technologies of the World Wide Web



# String manipulation in R

## What is string manipulation?

- processing of string (character, text) data
- important operations: extraction of text patterns, data tidying, preparation of text corpora for statistical text processing
- web data often is text data!

## String manipulation with R

- base R provides basic string manipulation functionality but not a very consistent syntax
- comfortable string processing with Hadley Wickham's **stringr** package

# String manipulation in R with the `stringr` package

Function	Description	Output
<i>Functions using regular expressions</i>		
<code>str_extract()</code>	Extracts first string that matches pattern	Character vector
<code>str_extract_all()</code>	Extracts all strings that match pattern	List of character vectors
<code>str_locate()</code>	Return position of first pattern match	Matrix of start/end positions
<code>str_locate_all()</code>	Return positions of all pattern matches	List of matrices
<code>str_replace()</code>	Replaces first pattern match	Character vector
<code>str_replace_all()</code>	Replaces all pattern matches	Character vector
<code>str_split()</code>	Split string at pattern	List of character vectors
<code>str_split_fixed()</code>	Split string into fixed number of pieces	Matrix of character vectors
<code>str_detect()</code>	Detect pattern in string	Boolean vector
<code>str_count()</code>	Count number of patterns in string	Numeric vector
<i>Further useful functions</i>		
<code>str_sub()</code>	Extract strings by position	Character vector
<code>str_subset()</code>	Extract strings for which condition applies	Character vector
<code>str_length()</code>	Length of string	Numeric vector
<code>str_trim()</code>	Discard string padding	Character vector

## **Useful functions**

---

# Regular expressions in R

Again, our example string:

R code —

---

```
1 example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

---

end

# String manipulation in R

"1. A small sentence. - 2. Another tiny sentence."

## String localization

R code

```
2 str_locate(example.obj, "small")
```

	start	end
[1,]	6	10

end

## Substring extraction

R code

```
3 str_sub(example.obj, start = 6, end = 10)
```

[1] "small"

end

# String manipulation in R

"1. A small sentence. - 2. Another tiny sentence."

## String replacement

R code

```
4 str_replace(example.obj, pattern = "tiny", replacement = "huge")
[1] "1. A small sentence. - 2. Another huge sentence."
```

end

## String splitting

R code

```
5 unlist(str_split(example.obj, "-"))
[1] "1. A small sentence. "      " 2. Another tiny sentence."
```

end

# String manipulation in R

## Manipulation of several elements

- until this point we applied functions to vectors of length one
- now: apply functions to multi-element vectors

Example object with several elements:

R code

---

```
6 (char.vec <- c("this", "and this", "and that"))
[1] "this"      "and this"   "and that"
```

end

---

## String detection

R code

---

```
7 str_detect(char.vec, "this")
[1] TRUE  TRUE FALSE
```

end

## String counting

R code

```
8 str_count(char.vec, "this")
[1] 1 1 0
9 str_count(char.vec, "\\w+")
[1] 1 2 2
10 str_length(char.vec)
[1] 4 8 8
```

end

## String subsetting

R code

```
11 str_subset(char.vec, "this")
[1] "this"      "and this"
```

end

## String joining

R code

---

```
12 str_c("text", "manipulation", sep = " ")
[1] "text manipulation"
13 cat(str_c(char.vec, collapse = "\n"))
this
and this
and that
14 str_c("text", c("manipulation", "basics"), sep = " ")
[1] "text manipulation" "text basics"
```

---

end

... but you might want prefer to stick to good old `paste()` and `paste0()`.

## Approximate matching

- Matching of approximately equal strings, (e.g., “Beyoncé” and “Beyonce”)
- we could program naïve matching algorithms using regex
- better: use of more powerful algorithms, e.g. Levenshtein distance
- `agrep()` function in base R
- for more functionality see `stringdist` package

R code

```
15  agrep("Barack Obama", "Barack H. Obama", max.distance = list(all = 3))
[1] 1
16  agrep("Barack Obama", "Michelle Obama", max.distance = list(all = 3))
integer(0)
```

end

## **Summary**

---

# Summary

- being able to manipulate string data in R is very useful when you want to automate processing web data
- there are base R commands for string manipulation, but the **stringr** package provides many useful functions with a consistent syntax
- if you need more, check out the even more powerful **stringi** package



Source:  
<http://kevingleong.blogspot.de/2011/01/string-manipulation-exercises.html>

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Inspecting the HTML Tree

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **Browsing vs. scraping**

---

# Browsing vs. scraping the web

## Using your browser to access webpages

1. you click on a link, enter a URL, etc.
2. browser/your machine sends request to server that hosts website
3. server returns resource (often an HTML document)
4. browser interprets HTML and renders it nicely



## Using R to access webpages

1. you manually specify a resource
2. R/your machine sends request to server
3. server returns resource
4. R parses HTML but does not render it
5. it's up to you to tell R what content to extract



# Interacting with your browser

## On web browsers

- modern browsers are complex pieces of software that take care of multiple operations while you browse the web
- common operations: retrieve resources, render and display information, provide interface for user-webpage interaction
- although your goal is to automate web data retrieval, your browser is an important tool in web scraping workflow



## The use of browsers for web scraping

- give you an intuitive impression of the architecture of a webpage
- allow you to inspect the source code
- let you construct XPath/CSS selector expressions with plugins
- render dynamic web content (JavaScript interpreter)

## A note on browser differences

- inspecting the source code (as shown on the following slides) works more or less identically in **Chrome** and **Firefox**
- in **Safari**, go to → **Preferences**, then → **Advanced** and select "Show Develop menu in menu bar". This unlocks the "Show Page Source" option and the Web Developer Tools

## **Inspecting the HTML source code**

---

# Inspecting the HTML source code

## Example

- browser: Google Chrome
- source: [https://en.wikipedia.org/wiki/List\\_of\\_tallest\\_buildings](https://en.wikipedia.org/wiki/List_of_tallest_buildings)

The screenshot shows a web browser window with the URL [https://en.wikipedia.org/w/index.php?title=List\\_of\\_tallest\\_buildings&oldid=910803020](https://en.wikipedia.org/w/index.php?title=List_of_tallest_buildings&oldid=910803020). The page title is "List of tallest buildings". On the left, there is a sidebar with a "Table of contents" section containing 12 items, each with a small icon and text. The main content area displays the heading "List of tallest buildings" and a sub-section titled "Ranking criteria and alternatives". A large image of the Burj Khalifa is visible on the right side of the page.

# Inspecting the HTML source code

## Example

- browser: Google Chrome
- source: [https://en.wikipedia.org/wiki/List\\_of\\_tallest\\_buildings](https://en.wikipedia.org/wiki/List_of_tallest_buildings)
- right-click on page



# Inspecting the HTML source code

## Example

- browser: Google Chrome
- source: [https://en.wikipedia.org/wiki/List\\_of\\_tallest\\_buildings](https://en.wikipedia.org/wiki/List_of_tallest_buildings)
- right-click on page
- select "View Page Source"



# Inspecting the HTML source code

## Example

- HTML (and JavaScript) code can be ugly...



# Inspecting the HTML source code

## Example

- HTML (and JavaScript) code can be ugly...
- but looking more closely, we can find the displayed information

The screenshot shows a browser window with the URL [www.nytimes.com/2012/02/01/us/police-shootings-in-black-communities.html](http://www.nytimes.com/2012/02/01/us/police-shootings-in-black-communities.html). The developer tools' element inspector is open, displaying the DOM structure of the page. A specific element, a `<div>` with the class `story-body__inner`, is selected and highlighted in red. This element contains several paragraphs of text, which are also highlighted in red, indicating they are the source of the visible content on the page.

## **Inspecting the live HTML tree**

---

# Inspecting individual elements and the live HTML tree

## Example

- another way of inspecting the HTML code is to use the **Web Developer Tools**
- to that end, do:
  - right-click on element of interest
  - select "Inspect"



# Inspecting individual elements and the live HTML tree

## Example

- the Web Developer Tools window pops up
- corresponding part in the HTML tree is highlighted
- you can hover over other parts of the code to get an instant view of the page at that position
- click on arrows to expand/hide tags



## **Summary**

---

# Summary

## When to inspect the complete page source

- check whether data is in static source code
- for small HTML files: understand structure
- count tables

## When to inspect individual elements using the Web Inspector Tools

- almost always
- particularly useful to construct XPath/CSS selector expressions
- to monitor dynamic changes in the DOM tree (see later)



Source: <http://watershedcreative.com/naked/html-tree.html>

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

XPath, Part I

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **Accessing the HTML tree with R**

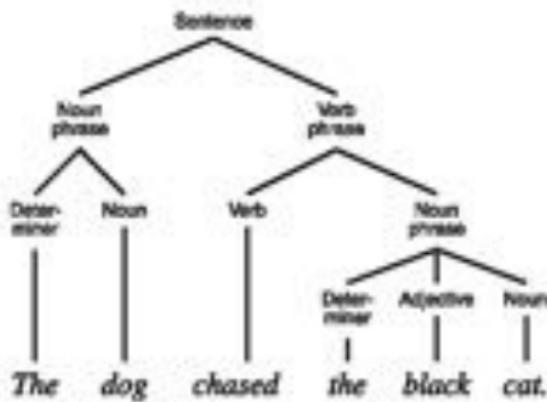
---

# Accessing the HTML tree with R

- HTML documents are human-readable
- HTML tags structure the document
- **web user perspective:** the browser interprets the code and renders the page
- **web scraper perspective:** use the tags to locate information; document has to be parsed first

## Parsing

Parsing originally describes the syntactic analysis of text according to grammatical rules; analysis of the relationship between single parts of text. In programming, the input has to be interpreted (e.g., by R) to process the command.



## Tools

- the `xml2` package allows us to parse XML-style documents
- the `rvest` package, which we will mainly use for scraping, wraps the `xml2` package, so we rarely have to load it manually
- HTML is a "flavor" of XML, so we can use the package to parse HTML
- one high-level function: `read_html()`
- `read_html()` represents the HTML in a list-style fashion
- we could also import HTML files via `readLines()`, but this is not parsing—the document's structure is not retained

# HTML parsing with R

## Parsing a website is straightforward

R code

---

```
1 library(rvest)  
2 parsed_doc <- read_html("https://google.com")
```

---

end

## Functions to inspect the parsed document - better use the browser instead

R code

---

```
3 html_structure(parsed_doc)  
4 as_list(parsed_doc)
```

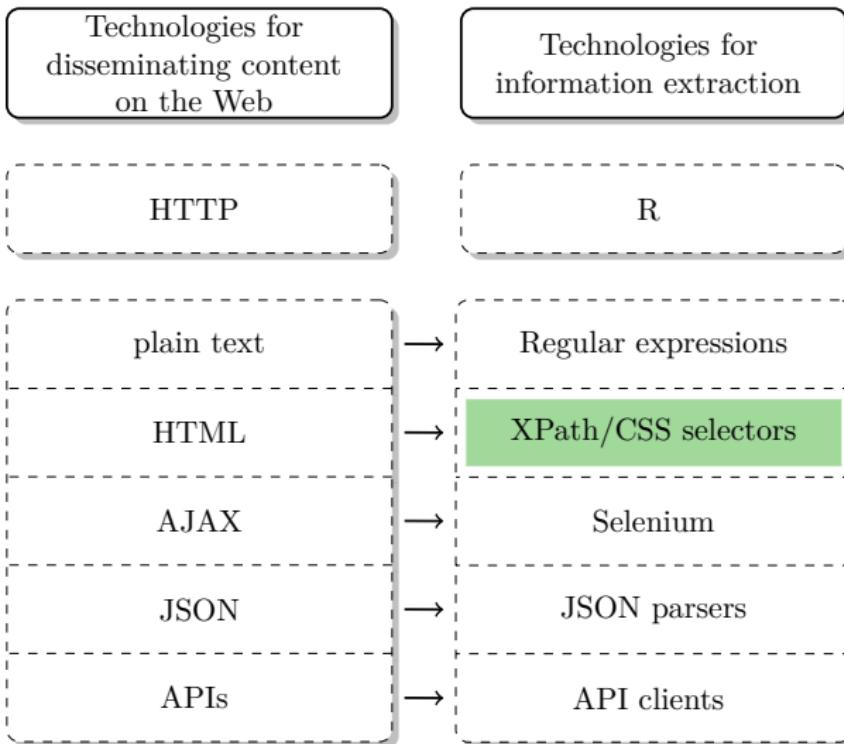
---

end

# XPath

---

# Technologies of the World Wide Web



# What's XPath?

## Definition

- XML Path language, a W3C standard
- query language for XML-based documents (→ HTML)
- access node sets and extract content

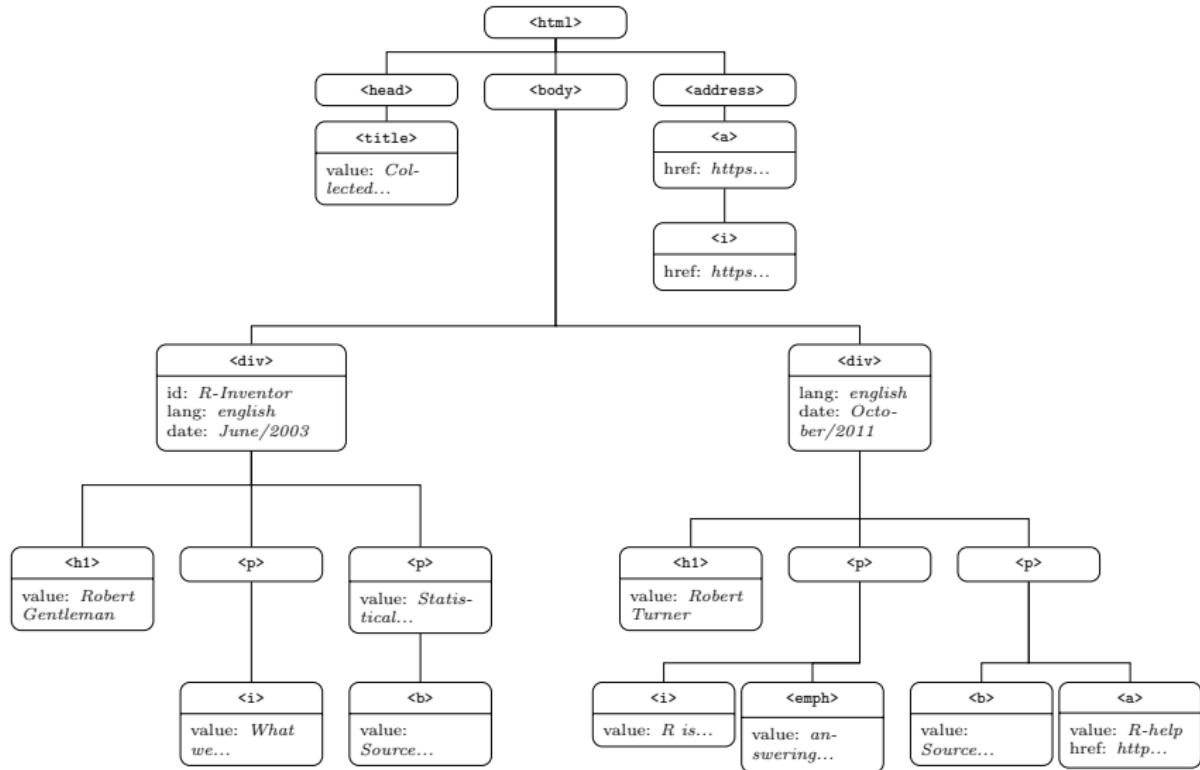
## Why XPath for web scraping?

- source code of webpages (HTML) structures both layout and content
- not only content, but context matters!
- enables us to extract content based on its location in the document and (usually) regardless of its shape

# Example

```
1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
2 <html> <head>
3 <title>Collected R wisdoms</title>
4 </head>
5 <body>
6 <div id="R Inventor" lang="english" date="June/2003">
7   <h1>Robert Gentleman</h1>
8   <p><i>'What we have is nice, but we need something very different'</i></p>
9   <p><b>Source: </b>Statistical Computing 2003, Reisenburg</p>
10 </div>
11 <div lang="english" date="October/2011">
12   <h1>Rolf Turner</h1>
13   <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering a
14     request for automatic generation of 'data from a known mean and 95% CI'</
15     emph></p>
16   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</
17     a></p>
18 </div>
19 </body>
20 <address><a href="http://www.r-datacollection.com"><i>The book homepage</i><a/></
21   address>
22 </html>
```

# Example



## Applying an XPath expression in R

- load package `rvest`
- parse document with `read_html()`
- query document with XPath expression using `html_nodes()`
- `rvest` can process XPath queries as well as CSS selectors
- in this course, we'll focus on XPath

R code

```
5 library(rvest)
6 parsed_doc <- read_html("../materials/fortunes.html")
7 html_nodes(parsed_doc, xpath = "//div[last()]/p/i")
{xml_nodeset (1)}
[1] <i>'R is wonderful, but it cannot work magic'</i>
```

end

# **The grammar of XPath**

---

# Grammar of XPath

## Basic rules

1. we access nodes by writing down the hierarchical structure in the DOM that locates the node set of interest
2. a sequence of nodes is separated by slash symbols
3. the easiest localization of a node is given by the absolute path (but often not the most efficient one!)
4. apply XPath on document in R with the `html_nodes()` function

R code

```
8 html_nodes(parsed_doc, xpath = "//div[last()]/p/i")
{xml_nodeset (1)}
[1] <i>'R is wonderful, but it cannot work magic'</i>
```

end

## Absolute vs. relative paths

- absolute paths start at the root node and follow the whole way down to the target node (with simple slashes, '/')
- relative paths skip nodes (with double slashes, '//')

R code

---

```
9 html_nodes(parsed_doc, xpath = "/html/body/div/p/i")
{xml_nodeset (2)}
[1] <i>'What we have is nice, but we need something very different'</i>
[2] <i>'R is wonderful, but it cannot work magic'</i>

10 html_nodes(parsed_doc, xpath = "//body//p/i")
{xml_nodeset (2)}
[1] <i>'What we have is nice, but we need something very different'</i>
[2] <i>'R is wonderful, but it cannot work magic'</i>
```

---

end

## When to use absolute, when relative paths?

- relative paths faster to write
- relative paths often more comprehensive (but less robust)
- relative paths consume more computing time, as the whole tree has to be parsed, but this is usually of less relevance for reasonably small documents

R code

---

```
11 html_nodes(parsed_doc, xpath = "//i")
{xml_nodeset (3)}
[1] <i>'What we have is nice, but we need something very different'</i>
[2] <i>'R is wonderful, but it cannot work magic'</i>
[3] <i>The book homepage</i>
```

---

end

## Wildcard operator

- meta symbol \*
- matches any node
- works only for one arbitrary node
- far less important than wildcards in regular expressions

R code

---

```
12 html_nodes(parsed_doc, xpath = "/html/body/div/*/i")
  {xml_nodeset (2)}
[1] <i>'What we have is nice, but we need something very different'</i>
[2] <i>'R is wonderful, but it cannot work magic'</i>
13 # this does not work:
14 html_nodes(parsed_doc, xpath = "/html/body/*/i")
  {xml_nodeset (0)}
```

---

end

## Navigational operators ‘.’ and ‘..’

- . accesses nodes at the same level ('self axis')
- useful when working with predicates
- .. accesses nodes at a higher hierarchical level

R code

---

```
15 html_nodes(parsed_doc, xpath = "//title/..")
{xml_nodeset (1)}
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=
...

```

---

end

## Pipe operator

- combines several paths

R code —

```
16 html_nodes(parsed_doc, xpath = "//address | //title")
{xml_nodeset (2)}
[1] <title>Collected R wisdoms</title>
[2] <address>\n<a href="http://www.r-datacollection.com"><i>The book hom
...
————— end
```

## **Summary**

---

# Summary

- XPath is a little language that lets you query specific parts of an XML-style document
- it has its own grammar (logic) and vocabulary
- in this session, you learned the basics of XPath
- in the next session, you will learn more advanced, powerful XPath expressions



Source: [https://commons.wikimedia.org/wiki/File:Mozie\\_Law\\_path\\_junction\\_-\\_geograph.org.uk\\_-\\_1131.jpg](https://commons.wikimedia.org/wiki/File:Mozie_Law_path_junction_-_geograph.org.uk_-_1131.jpg) (Andy Stephenson)

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

XPath, Part II

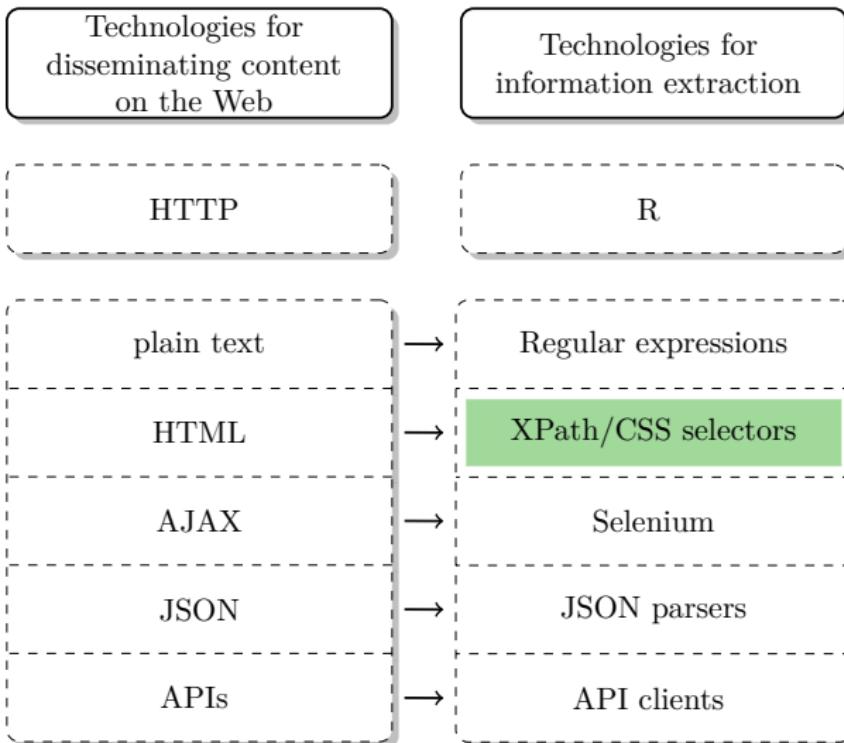
---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

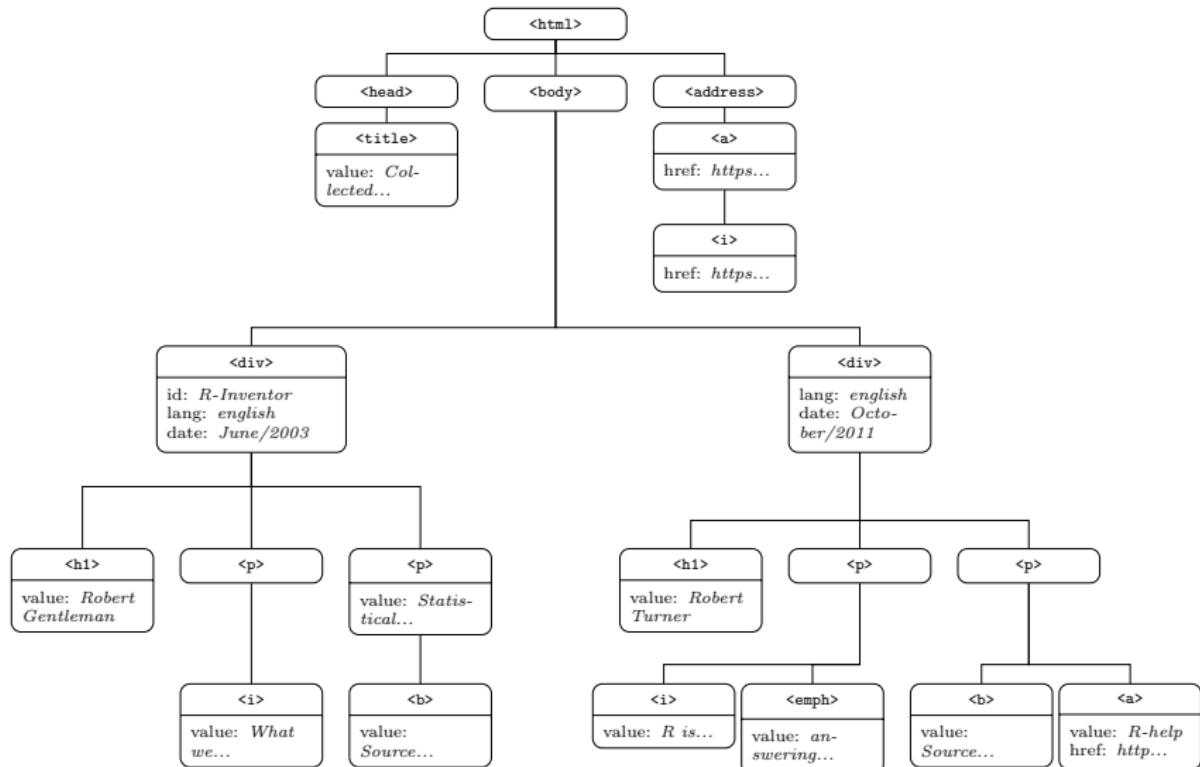
# Technologies of the World Wide Web



# Example

```
1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
2 <html> <head>
3 <title>Collected R wisdoms</title>
4 </head>
5 <body>
6 <div id="R Inventor" lang="english" date="June/2003">
7   <h1>Robert Gentleman</h1>
8   <p><i>'What we have is nice, but we need something very different'</i></p>
9   <p><b>Source: </b>Statistical Computing 2003, Reisenburg</p>
10 </div>
11 <div lang="english" date="October/2011">
12   <h1>Rolf Turner</h1>
13   <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering a
14     request for automatic generation of 'data from a known mean and 95% CI'</
15     emph></p>
16   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</
17     a></p>
18 </div>
19 </body>
20 <address><a href="http://www.r-datacollection.com"><i>The book homepage</i><a/></
21   address>
22 </html>
```

# Example



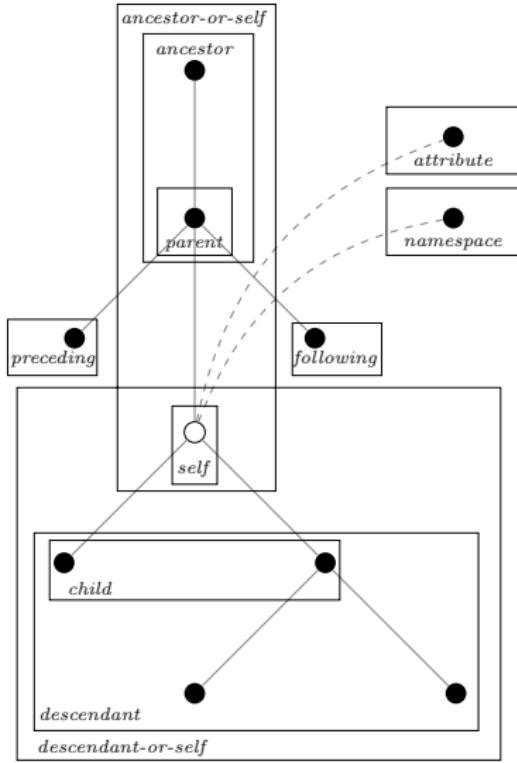
## **Node relations in XPath**

---

## 'Family relations' between nodes

- the tools learned so far are sometimes not sufficient to access specific nodes without accessing other, undesired nodes as well
- relationship statuses are useful to establish unambiguity
- can be combined with other elements of the grammar
- basic syntax: `node1/relation::node2`
- we describe *relation* of `node2` to `node1`
- `node2` is to be extracted—we **always** extract the node at the end

# Node relations in XPath



# Node relations in XPath

---

Axis name	Result
ancestor	all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	all ancestors of the current node and the current node itself
attribute	all attributes of the current node
child	all children of the current node
descendant	all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	all descendants of the current node and the current node itself
following	everything in the document after the closing tag of the current node
following-sibling	all siblings after the current node
namespace	all namespace nodes of the current node
parent	the parent of the current node
preceding	all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
preceding-sibling	all siblings before the current node
self	the current node

---

# Node relations in XPath

Example: access the `<div>` nodes that are ancestors to an `<a>` node:

R code

---

```
1 html_nodes(parsed_doc, xpath = "//a/ancestor::div")
{xml_nodeset (1)}
[1] <div lang="english" date="October/2011">\n    <h1>Rolf Turner</h1>\n    ...

```

---

end

Another example: Select all `<h1>` nodes that precede a `<p>` node:

R code

---

```
2 html_nodes(parsed_doc, xpath = "//p/preceding-sibling::h1")
{xml_nodeset (2)}
[1] <h1>Robert Gentleman</h1>
[2] <h1>Rolf Turner</h1>

```

---

end

# Predicates

---

## Predicates

- Conditions based on a node's features (`true/false`)
- applicable to a variety of features: name, value, attribute
- basic syntax:

*node [predicate]*

# Commands in predicates

Function	Returns...
<code>name(&lt;node&gt;)</code>	name of <code>&lt;node&gt;</code> or the first node in a node set
<code>text(&lt;node&gt;)</code>	value of <code>&lt;node&gt;</code> or the first node in a node set
<code>@attribute</code>	value of a node's <i>attribute</i>
<code>string-length(str1)</code>	length of <code>str1</code> . If there is no string argument, it length of the string value of the current node
<code>translate(str1, str2, str3)</code>	<code>str1</code> by replacing the characters in <code>str2</code> with the characters in <code>str3</code>
<code>contains(str1,str2)</code>	TRUE if <code>str1</code> contains <code>str2</code> , otherwise FALSE
<code>starts-with(str1,str2)</code>	TRUE if <code>str1</code> starts with <code>str2</code> , otherwise FALSE
<code>substring-before(str1,str2)</code>	start of <code>str1</code> before <code>str2</code> occurs in it
<code>substring-after(str1,str2)</code>	remainder of <code>str1</code> after <code>str2</code> occurs in it
<code>not(arg)</code>	TRUE if the Boolean value is FALSE, and FALSE if the boolean value is TRUE
<code>local-name(&lt;node&gt;)</code>	name of the current <code>&lt;node&gt;</code> or the first node in a node set – without the namespace prefix
<code>count(&lt;node&gt;)</code>	count of a nodeset <code>&lt;node&gt;</code>
<code>position(&lt;node&gt;)</code>	index position of <code>&lt;node&gt;</code> that is processed
<code>last()</code>	number of items in the processed node list <code>&lt;node&gt;</code>

# Predicates

## Numeric predicates indicate positions, counts, etc.

Example: Select all first `<p>` nodes that are children of a `<div>` node:

R code

```
3 html_nodes(parsed_doc, xpath = "//div/p[position()=1]")
{xml_nodeset (2)}
[1] <p><i>'What we have is nice, but we need something very different'</i>
...
[2] <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answe
...

```

---

end

Further examples:

R code

```
4 html_nodes(parsed_doc, xpath = "//div/p[last()-1]")
5 html_nodes(parsed_doc, xpath = "//div[count(./*)>2]")
6 html_nodes(parsed_doc, xpath = "//*[string-length(text())>50]")

```

end

## Textual predicates

- describe text features
- applicable on: node name, content, attributes, attribute values
- XPath 2.0 supports (simplified) regex, however, R (the **rvest** package and the underlying **xml2** package) does not support it

Example: Select all `<div>` nodes that contain an attribute named '`October/2011`':

R code

---

```
7 html_nodes(parsed_doc, xpath = "//div[@date='October/2011'])  
{xml_nodeset (1)}  
[1] <div lang="english" date="October/2011">\n    <h1>Rolf Turner</h1>\n    ...
```

---

end

## Partial matching

- rudimentary string matching
- 'content contains' (`contains()`), 'content begins with' (`starts-with()`), 'content ends with' (`ends-with()`), 'content contains after split' (`substring-after()`)

R code

---

```
8 html_nodes(parsed_doc, xpath = "//*[contains(text(), 'magic')]")
{xml_nodeset (1)}
[1] <i>'R is wonderful, but it cannot work magic'</i>
```

end

---

Further examples:

R code

---

```
9 html_nodes(parsed_doc, xpath = "//div[starts-with(./@id, 'R')]")
10 html_nodes(parsed_doc, xpath = "//div[substring-after(./@date, '/') 
='2003']//i")
```

end

## **Content extraction**

---

# Extraction of text and attributes

## Extraction

- until now: query of complete nodes
- common scenario: only parts of the node are interesting, e.g., content (value)
- additional extraction operations from a selected node set possible with additional extractor functions

Function	Argument	Return value
<code>html_text</code>		node value
<code>html_attr</code>	<code>name</code>	node attribute
<code>html_attrs</code>		(all) node attributes
<code>html_name</code>	<code>trim</code>	node name
<code>html_children</code>		node children

# Extraction of node elements

## Values/text

R code

```
11 html_nodes(parsed_doc, xpath = "//title") %>% html_text()  
[1] "Collected R wisdoms"
```

end

## Attributes

R code

```
12 html_nodes(parsed_doc, xpath = "//div") %>% htmlAttrs()  
[[1]]  
      id          lang         date  
"R Inventor"    "english"   "June/2003"  
  
[[2]]  
      lang         date  
"english"   "October/2011"
```

end

# Extraction of node elements

## Attribute values

R code —

```
13 html_nodes(parsed_doc, xpath = "//div") %>% html_attr("lang")
[1] "english" "english"
```

— end

## **A silver lining**

---

# Do I really have to construct XPath expressions all by my own?

No!



## XPath creator tools

- *Selectorgadget*: <http://selectorgadget.com/>. Browser plugin that constructs XPath statements via a point-and-click approach. The generated expressions are not always efficient though
- *Web Developer Tools*: internal browser functionality which return XPath statements for selected nodes
- you will learn how to use these tools in upcoming sessions

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Scraping HTML Tables

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# HTML tables are everywhere

Purchased Equipments (June, 2006)			
Item Num#	Item Picture	Item Description	Price
		Shipping Handling, Installation, etc.	Expense
1.		IBM Clone Computer	\$ 400.00
		Shipping Handling, Installation, etc.	\$ 20.00
2.		1GB RAM Module for Computer	\$ 50.00
		Shipping Handling, Installation, etc.	\$ 14.00

Category	Product	Score	Count	Percent	Revenue	Profit	Margin
Electronics	Keyboard	80	1,000	10%	\$100,000	\$10,000	10%
Electronics	Computer Monitor	80	1,000	10%	\$100,000	\$10,000	10%
Electronics	Mouse	80	1,000	10%	\$100,000	\$10,000	10%
Electronics	Keyboard	80	1,000	10%	\$100,000	\$10,000	10%
Electronics	Mouse	80	1,000	10%	\$100,000	\$10,000	10%

Building #	Building	City	Country	Year	Area (sq ft)	Rooms	Occupants
10001	Academy City Building	New York City	United States	2000	100000	100	Demolished by Fire in 1990
10002	Auditorium Building	Chicago	United States	1900	100000	100	Standing
10003	Auto Parts Store Building	New York City	United States	2000	100000	100	Demolished in 1980
10004	Philadelphia City Hall	Philadelphia	United States	1920	100000	100	Standing
10005	Brown Building	Seattle	United States	1900	100000	100	Demolished in 1990
10006	Capitol City Tower	Seattle	United States	1900	100000	100	Standing
10007	Microsoft Building	Seattle	United States	1900	100000	100	Standing
10008	All Red House	New York City	United States	1900	100000	100	Standing
10009	Blue House Building	New York City	United States	1900	100000	100	Standing
10010	Empire State Building	New York City	United States	1900	100000	100	Standing
10011	World Trade Center (North Tower)	New York City	United States	1900	100000	100	Damaged in 2001 in the September 11 attacks
10012	White House - Executive Office Building	Washington D.C.	United States	1900	100000	100	Standing
10013	Pentagon Tower	Arlington County	United States	1900	100000	100	Standing
10014	Nasa M.	Texas	United States	1900	100000	100	Standing
10015	Rockefeller	Rockefeller Center	United States	1900	100000	100	Standing

# Recall: HTML tables share the same basic syntax

## Syntax

- HTML tables are easy to spot in the wild
- just look for `<table>` tags

## Example

```
1 <table>
2   <tr> <th>Rank</th> <th>Nominal GDP</th> <th>Name</th> </tr>
3   <tr> <th></th> <th>(per capita, USD)</th> <th></th> </tr>
4   <tr> <td>1</td> <td>170,373</td> <td>Lichtenstein</td> </tr>
5   <tr> <td>2</td> <td>167,021</td> <td>Monaco</td> </tr>
6   <tr> <td>3</td> <td>115,377</td> <td>Luxembourg</td> </tr>
7   <tr> <td>4</td> <td>98,565</td> <td>Norway</td> </tr>
8   <tr> <td>5</td> <td>92,682</td> <td>Qatar</td> </tr>
9 </table>
```

# **Scraping HTML tables in R**

---

# Scraping HTML tables in R

## A neat `rvest` solution

- exactly because scraping tables is an easy and repetitive task, there is a dedicated function for it (literally from the documentation):

```
html_table(x, header = NA, trim = TRUE, fill = FALSE, dec  
          = ".")
```

---

<code>x</code>	A node, node set or document
<code>header</code>	Use first row as header? If <code>NA</code> , will use first row if it consists of <code>&lt;th&gt;</code> tags
<code>trim</code>	Remove leading and trailing whitespace within each cell?
<code>fill</code>	If <code>TRUE</code> , automatically fill rows with fewer than the maximum number of columns with <code>NAs</code>
<code>dec</code>	The character used as decimal mark

# Example

## Example

- source: [https://en.wikipedia.org/wiki/List\\_of\\_human\\_spaceflights](https://en.wikipedia.org/wiki/List_of_human_spaceflights)

The screenshot shows a Wikipedia article titled "List of human spaceflights". The page includes a sidebar with navigation links like "Main page", "Recent changes", "Random page", and "Help", as well as a search bar at the top. The main content features a heading "List of human spaceflights" with a sub-section "Past flights: See also: astronauts". Below this is a paragraph about the history of human spaceflight, mentioning the first orbital flight by Yuri Gagarin in 1961 and the first American in space by John Glenn in 1962. It also notes the first spacewalk by Alexei Leonov in 1965 and the first landing on the Moon by Neil Armstrong in 1969. A sidebar on the right contains a "Summary" section with a table showing the number of flights per decade from 1960 to 2010, and a chart showing the total number of flights over time.

Decade	1960s	1970s	1980s	1990s	2000s	Total
Flights	10	10	10	10	10	50
Astronauts	10	10	10	10	10	50

Summary (50)

Decade	1960s	1970s	1980s	1990s	2000s	Total
Flights	10	10	10	10	10	50
Astronauts	10	10	10	10	10	50

Flights by Decade (1960–2010)

Number of flights per decade from 1960 to 2010, including international partners. As of 2010, there have been 50 recorded spaceflights that resulted in 100 or more days in orbit.

# Example

## Example

- source: [https://en.wikipedia.org/wiki/List\\_of\\_human\\_spaceflights](https://en.wikipedia.org/wiki/List_of_human_spaceflights)
- not entirely clean table: some cells empty, images, links

Screenshot of a Wikipedia page titled "List of human spaceflights". The page contains a table with data about human spaceflights.

**Table Data:**

Decade	Count	First flight	Last flight
1960s	10	1961-04-12 Gagarin	1969-12-14 Shephard
1970s	13	1971-04-19 Soyuz 10	1978-08-24 Skylab 4
1980s	10	1981-04-12 Columbia	1990-07-13 Mir 17
1990s	10	1991-01-19 Endeavour	1999-06-02 Cassiopeia
2000s	10	2001-01-16 Discovery	2009-07-21 Shuttle-Atlantis
2010s	10	2011-05-08 Shuttle-Atlantis	2015-07-08 Tiangong 2
Total	83		

**Summary:**

- 83 total flights
- 10 flights per decade
- First flight: 1961-04-12 (Gagarin)
- Last flight: 2015-07-08 (Tiangong 2)

**Image:** A small image of a rocket launching, labeled "Launch of Soyuz 10, 1971".

**Figure:** A bar chart showing the number of flights per decade: 1960s (~10), 1970s (~13), 1980s (~10), 1990s (~10), 2000s (~10), 2010s (~10). The chart is titled "Total of all human spaceflights" and includes a note: "Based on 17 international flights (1973-2014), including commercial launches (2008-2014) and private flights (2014-2015)".

## Example

## Example

- source: [https://en.wikipedia.org/wiki/List\\_of\\_human\\_spaceflights](https://en.wikipedia.org/wiki/List_of_human_spaceflights)
  - not entirely clean table: some cells empty, images, links
  - HTML code is straightforward



# Example

## Example

- source: [https://en.wikipedia.org/wiki/List\\_of\\_human\\_spaceflights](https://en.wikipedia.org/wiki/List_of_human_spaceflights)
- not entirely clean table: some cells empty, images, links
- HTML code is straightforward

```
<table class="wikitable" style="text-align:right;">
  <tbody>
    <tr></tr>
    <tr></tr>
    <tr>
      <td></td>
      <td>30</td>
      <td>8</td>
      <td></td>
      <td>38</td>
    </tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
  </tbody>
</table>
```

# Example

## Scraping the table with R

R code

```
1 library(rvest)
2 url_p <- read_html("https://en.wikipedia.org/wiki/List_of_human_
spaceflights")
3 tables <- html_table(url_p, header = TRUE, fill = TRUE)
4 spaceflights <- tables[[1]]
5 spaceflights
```

	Russia	Soviet Union	United States	China	Total	
1	1961-1970		16	25	NA	41
2	1971-1980		30	8	NA	38
3	1981-1990		*25	*38	NA	*63
4	1991-2000		20	63	NA	83
5	2001-2010		24	34	3	61
6	2011-2020		28	3	3	34
7	Total		*143	*171	6	*320

end

## **Summary**

---

# Summary

- scraping HTML tables with R is straightforward
- the high-level `html_table()` function is easy to use and generally robust
- sometimes, HTML tables are a bit more complex. You might encounter:
  - tables where cells span multiple rows
  - tables where headers are not in the first row
- in such cases, you might want to check out the `htmltab` package, which is more complex to use but also provides more flexibility



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Using SelectorGadget

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# **SelectorGadget**

---

## Scraping webpages

- the most cumbersome part of web scraping (data tidying aside) is the construction of XPath expressions that match the components of a page you want to extract
- it will take a couple of scraping projects until you'll truly have mastered XPath

## A much-appreciated helper

- **SelectorGadget** is a JavaScript browser plugin that constructs XPath statements (or CSS selectors) via a point-and-click approach
- it is available here: <http://selectorgadget.com/> (there's also a Chrome extension)
- the tool is magic and you will love it

## **What does SelectorGadget do?**

---

# What does SelectorGadget do?

## On the surface

1. you activate the tool on any webpage you want to scrape
2. based on your selection of components, the tool learns about your desired components and generates an XPath expression (or CSS selector) for you

## Under the hood

- based on your selection(s), the tool looks for similar elements on the page
- the underlying algorithm, which draws on Google's diff-match-patch libraries, focuses on CSS characteristics, such as tag names and `<div>` and `<span>` attributes

# Installation and use

## Installation

Go to <http://selectorgadget.com/> and drag the link provided at the end of the page ("Or drag this link...") to your bookmark bar.

## Use

1. Open the webpage you want to scrape
2. Click on the SelectorGadget bookmarklet
3. Click on the elements you want to extract. Manually selected elements will turn **green**. Elements that match the selector will be highlighted **yellow**. De-select the elements you do not want to extract - they will turn **red**.
4. Repeat step 3 until the marked set of elements matches the set you want to extract
5. Click on the "XPath" button in the console
6. Copy the generated XPath expression and process it with **rvest**

## **Example**

---

# Example

## Example

- source: <https://nytimes.com>



# Example

## Example

- source: <https://nytimes.com>
- goal: scrape all headlines



# Example

## Example

1. click on SelectorGadget bookmarklet (not shown)
2. click on first headline



# Example

## Example

1. click on SelectorGadget bookmarklet (not shown)
2. click on first headline



# Example

## Example

1. click on SelectorGadget bookmarklet (not shown)
2. click on first headline
3. all relevant headlines selected



# Example

## Example

1. click on SelectorGadget bookmarklet (not shown)
2. click on first headline
3. all relevant headlines selected
4. click on "XPath"
5. copy proposed expression



# Example

## Now, we switch to R

R code

```
1 library(rvest)
2 url_p <- read_html("https://www.nytimes.com")
3 headlines <- html_nodes(url_p, xpath = '//*[@contains(concat( " ", @class
 , " " ), concat( " ", "story-heading", " " ))]//a') # we use single
quotation marks here to wrap around the expression!
4 headlines_raw <- html_text(headlines)
5 head(headlines_raw)
[1] "Fresh Worries on Russia for Trump's Weary Defenders"
[2] "F.B.I. Confirms Inquiry on Trump Team's Russia Ties"
[3] "Takeaways From the Hearing "
[4] "\n      Trump and the Russians: Links? No Links?"
[5] "G.O.P. Responds by Changing Subject"
[6] "U.S. Limits Devices on Foreign Airlines From 8 Countries"
```

end

# Example

## Let's clean the data

R code —

```
6 headlines_clean <- headlines_raw %>% str_replace_all("\n", "") %>% str_
trim()

7 length(headlines_clean)
[1] "Fresh Worries on Russia for Trump's Weary Defenders"
[2] "F.B.I. Confirms Inquiry on Trump Team's Russia Ties"
[3] "Takeaways From the Hearing"
[4] "Trump and the Russians: Links? No Links?"
[5] "G.O.P. Responds by Changing Subject"
[6] "U.S. Limits Devices on Foreign Airlines From 8 Countries"

8 str_detect(headlines_clean, "Trump") %>% table()
FALSE    TRUE
 133      12
```

end

## **Summary**

---

# Summary

So why did I bother you with learning XPath at all?

## Caveats

- SelectorGadget is not perfect. Sometimes, the algorithm will fail
- starting from a different element sometimes (but not always!) helps
- often the generated expressions are unnecessarily complex and therefore difficult to debug
- in my experience, SelectorGadget works 70-80% of the times when scraping from static webpages
- you are also prepared for the remaining 20-30%!



Source: [http://inspectorgadget.wikia.com/wiki/File:Inspector\\_Gadget\\_Thinking.png](http://inspectorgadget.wikia.com/wiki/File:Inspector_Gadget_Thinking.png)  
(DANTHEMAN123)

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

The Scraping Workflow

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

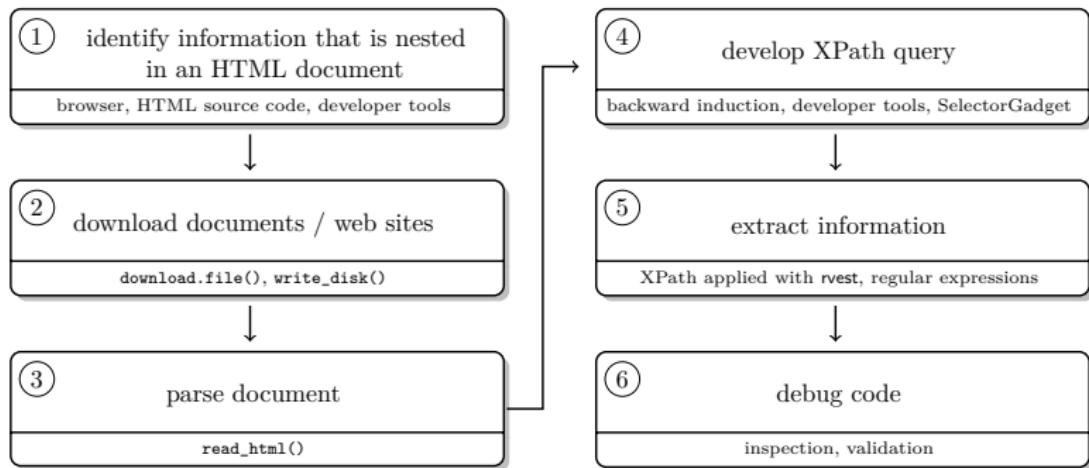
## You have learned the main tools necessary to scrape static webpages with R

1. you are able to inspect HTML pages in your browser using the web developer tools
2. you are able to parse HTML into R with `rvest`
3. you are able to speak XPath
4. you are able to apply XPath expressions with `rvest`
5. you are able to tidy web data with your R skills and regular expressions

## **The scraping workflow**

---

# The scraping workflow



# Downloading HTML files

## Stay modest when accessing lots of data

- content on the web is publicly available, ...
- but accessing the data causes server traffic
- stay polite by querying resources as sparsely as possible

## Two easy-to-implement practices

1. do not bombard the server with requests—and if you have to, do at a reasonable speed
2. download HTML files first, then parse

# Downloading HTML files

R code

```
1 for (i in 1:length(list_of_urls)) {  
2     if (!file.exists(paste0(folder, file_names[i]))) {  
3         download.file(list_of_urls[i], destfile = paste0(folder, file_  
names[i]))  
4         Sys.sleep(runif(1, 1, 2))  
5     }  
6 }
```

end

## Looping over a list of URLs

- `!file.exists()` checks whether a file does not yet exist in the local folder
- `download.file()` downloads the file to a folder; file name has to be specified
- `Sys.sleep()` suspends the execution of R code for a given time interval. Here: random interval between 1 and 2 seconds

## Don't be a phantom

- downloading massive amounts of data may arouse attention from server administrators
- assuming that you've got nothing to hide, you should stay identifiable beyond your IP address

## Two easy-to-implement practices

1. personally get in touch with website owners
2. use HTTP header fields `From` and `User-Agent` to provide information about yourself

# Staying identifiable

R code —

```
7 url <- "http://a-totally-random-website.com"
8 session <- html_session(url, add_headers(From = "my@email.com", `User-
Agent` = R.Version()$version.string)))
9 headlines <- session %>% html_nodes(xpath = "p//a") %>% html_text()

```

end

## The code snippet explained

- `rvest`'s `html_session()` creates a session object that responds to HTTP and HTML methods
- here, we provide our email address and the current R version as User-Agent information
- this will pop up in the server logs—the webpage administrator has the chance to easily get in touch with you

## **Summary**

---

# Summary

- the basic scraping workflow with R is straightforward
- with great power comes great responsibility: stay polite on the web when scraping lots of data!
- more complexity is added when you want to gather data from multiple websites, or when dynamic elements such as forms or JavaScript content is involved
- we will consider such cases in upcoming sessions



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Scraping Multiple Pages

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## An advanced scraping scenario

---

# Motivation

---

- until now, the toy examples were limited to single HTML pages
- often, we want to scrape data from multiple pages
- in such scenarios, automating the scraping process becomes **really** powerful
- also, the principles of polite scraping are more relevant

# The scenario

**Goal:** examine download statistics of articles of the Journal of Statistical Software

- download HTML pages
- extract bibliometrical information

## Tasks:

- identify relevant resources on  
<http://www.jstatsoft.org/>
- download HTML pages
- import them into R
- extract information via XPath



# **Scraping multiple pages with R**

---

# Step 1: Inspect the source

## Procedure

- source:  
<http://www.jstatsoft.org/>
- go to "Archives"



# Step 1: Inspect the source

## Procedure

- source:  
<http://www.jstatsoft.org/>
- go to "Archives"
- inspect the most recent volume



# Step 1: Inspect the source

## Procedure

- source:  
<http://www.jstatsoft.org/>
- go to "Archives"
- inspect the most recent volume
- inspect the first article

The screenshot shows a web browser displaying the *Journal of Statistical Software* website. The URL in the address bar is <http://www.jstatsoft.org/vol80/>. The page title is "Journal of Statistical Software". Below the title, it says "Vol 80 (2017)". A "Table of Contents" section is visible, listing several articles. The first article, "A New R Package for Computing the Beta Function and Related Quantities" by Michael M. Trager, is highlighted with a red rectangular box around its title and abstract. The sidebar on the right contains various navigation links and search fields.

# Step 1: Inspect the source

## Procedure

- source:  
<http://www.jstatsoft.org/>
- go to "Archives"
- inspect the most recent volume
- inspect the first article



# Step 1: Inspect the source

## Procedure

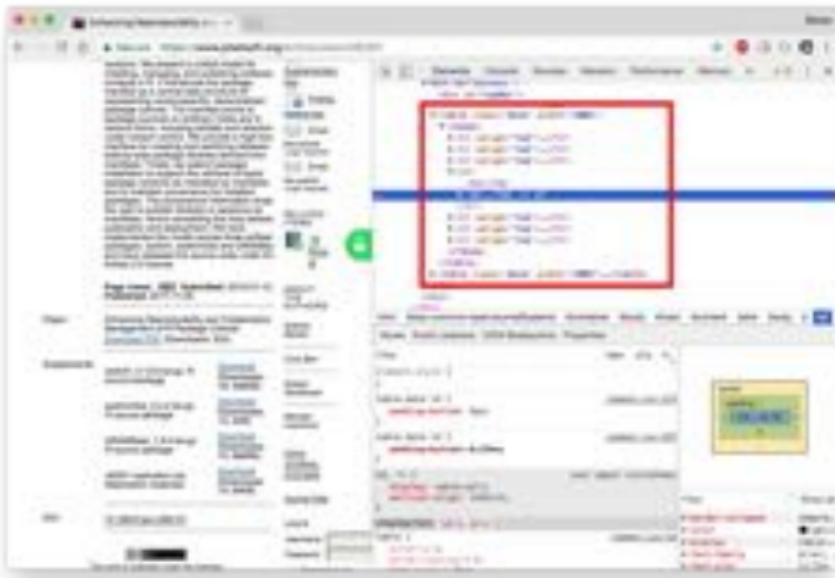
- source:  
<http://www.jstatsoft.org>
- go to "Archives"
- inspect the most recent volume
- inspect the first article
- inspect the page views element



# Step 1: Inspect the source

## Procedure

- source:  
<http://www.jstatsoft.org/>
- go to "Archives"
- inspect the most recent volume
- inspect the first article
- inspect the page views element
- it's in a table!



## Step 2: Develop a scraping strategy

### Observations

- getting the information out of the table will be straightforward
- this applies to all articles (check other articles on a sample basis)
- what we need is the set of **URLs leading to all articles**

### Inspecting the URLs

- the URL of the selected article looks as follows:  
<https://www.jstatsoft.org/article/view/v082i01>
- we find out that the final part, `v082i01`, always follows the same pattern:  
`v<volume number>i<issue number>`

## Step 2: Develop a scraping strategy

### Let's try to construct the list of URLs from scratch

R code

---

```
1 baseurl <- "http://www.jstatsoft.org/article/view/v"
2 volurl <- paste0("0", seq(1, 78, 1))
3 volurl[1:9] <- paste0("00", seq(1, 9, 1))
4 brurl <- paste0("0", seq(1, 9, 1))
5 urls_list <- paste0(baseurl, volurl)
6 urls_list <- paste0(rep(urls_list, each = 9), "i", brurl)
7 urls_list[1:5]
[1] "http://www.jstatsoft.org/article/view/v001i01"
[2] "http://www.jstatsoft.org/article/view/v001i02"
[3] "http://www.jstatsoft.org/article/view/v001i03"
[4] "http://www.jstatsoft.org/article/view/v001i04"
[5] "http://www.jstatsoft.org/article/view/v001i05"
8 names <- paste0(rep(volurl, each = 9), "_", brurl, ".html")
9 names[1:5]
[1] "001_01.html" "001_02.html" "001_03.html" "001_04.html" "001_05.html"
"
```

# Step 3: Download the files

## Set working directory

R code

```
10 tempwd <- ("data/jstatsoftStats")
11 dir.create(tempwd)
12 setwd(tempwd)
```

end

## Download pages

R code

```
13 folder <- "html_articles/"
14 dir.create(folder)
15 for (i in 1:length(urls_list)) {
16   if (!file.exists(paste0(folder, names[i]))) {
17     download.file(urls_list[i], destfile = paste0(folder, names[i]))
18     Sys.sleep(runif(1, 0, 1))
19   }
20 }
```

## Step 3: Download the files

### Check success

R code

---

```
21 list_files <- list.files(folder, pattern = "0.*")
22 list_files_path <- list.files(folder, pattern = "0.*", full.names = TRUE
  )
23 length(list_files)

[1] 666
```

---

end

# Step 4: Import files and parse out information

## Build loop

R code

---

```
24 authors <- character()
25 title <- character()
26 statistics <- character()
27 numViews <- numeric()
28 datePublish <- character()
29 for (i in 1:length(list_files_path)) {
30   html_out <- read_html(list_files_path[i])
31   table_out <- html_table(html_out, fill = TRUE)[[6]]
32   authors[i] <- table_out[1, 2]
33   title[i] <- table_out[2, 2]
34   statistics[i] <- table_out[4, 2]
35   numViews[i] <- statistics[i] %>% str_extract("[[:digit:]]+")
36   %>% as.numeric()
37   datePublish[i] <- statistics[i] %>% str_extract("[[:digit:]]{4}-[[:digit:]]{2}-[[:digit:]]{2}.\$")
38   %>% str_replace("\\.", "")
```

## Step 4: Import files and parse out information

### Inspect parsed data

R code

---

```
39 authors[1:3]
[1] "Ronald Barry"    "Jason Bond, George Michailides"    "Thomas Lumley"
40 title[1:2]
[1] "A Diagnostic to Assess the Fit of a Variogram Model to Spatial Data
"
[2] "Homogeneity Analysis in Xlisp-Stat"
41 numViews[1:3]
[1] 5835 3939 4379
```

---

end

### Construct data frame

R code

---

```
42 dat <- data.frame(authors = authors, title = title, numViews = numViews,
  datePublish = datePublish)
43 dim(dat)
```

## Step 5: Visualize data

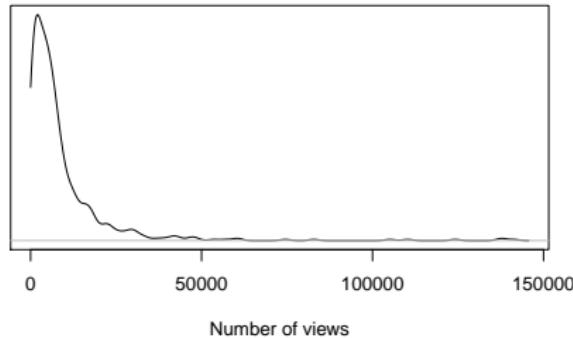
### Density plot of download statistics

R code —

```
44 plot(density(dat$numViews, from = 0), yaxt="n", ylab="", xlab="Number of  
views", main="Distribution of article page views in JStatSoft")
```

— end

**Distribution of article page views in JStatSoft**



## **Summary**

---

# Summary

- scraping data from multiple pages is no problem in R
- most of the brain work often goes into developing a scraping strategy and tidying the data, not into the actual downloading/scraping part
- scraping is also possible in even more complex scenarios, e.g., when HTML forms are involved or you have to take care of cookies or authentication
- this is beyond the scope of this course → check out the book for more applications



Source: Horia Varlan

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Dynamic Webpages

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **Static webpages**

---

# The simple world of static webpages

## Static webpages

- every user who visits the site gets the same content (unless the developer edits the source code)
- example: <https://www.jstatsoft.org>
- can contain "dynamic features", such as video or sound, but the page itself is not dynamic

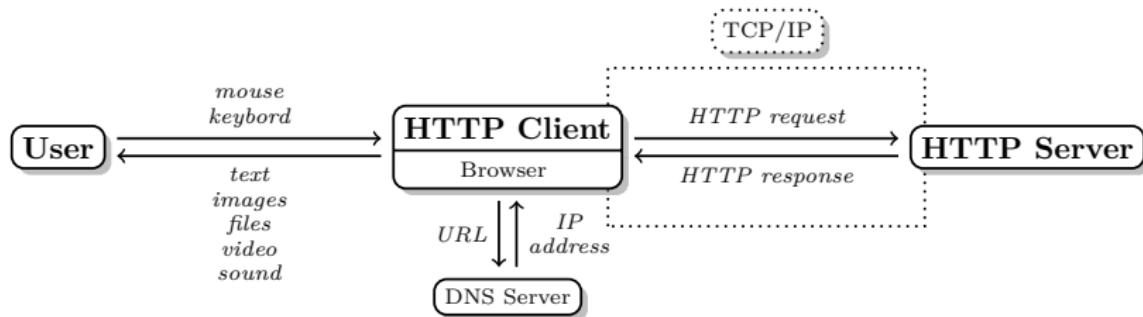


Screenshot of Amazon.com in  
1997

# A tiny bit of HTTP

## Client-server communication with HTTP

- HTTP, the **Hypertext Transfer Protocol**, is a stateless protocol
- no information is retained by either sender or receiver
- makes interaction with websites straightforward, but not very exciting



## Client-server communication with HTTP

### 1. Establishing connection

```
1 About to connect() to www.r-datacollection.com port 80 (#0)
2 Trying 173.236.186.125... connected
3 Connected to www.r-datacollection.com (173.236.186.125) port 80 (#0)
4 Connection #0 to host www.r-datacollection.com left intact
```

### 2. HTTP request

```
1 GET /index.html HTTP/1.1
2 Host: www.r-datacollection.com
3 Accept: */*
```

# A tiny bit of HTTP

## Client-server communication with HTTP

### 3. HTTP response

```
1 HTTP/1.1 200 OK
2 Date: Thu, 27 Feb 2014 09:40:35 GMT
3 Server: Apache
4 Vary: Accept-Encoding
5 Content-Length: 131
6 ...
7
8 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
9 <html> <head>
10 <title></title>
11 </head>
12 ...
```

### 4. Closing connection

```
1 Closing connection #0
```

# The "problem" of static webpages

## Static webpages reconsidered

- HTML/HTTP are used for static display of content → same content for every visitor
- in order to display dynamic content, they lack
  1. mechanisms to detect user behavior in the browser (and not only on the server)
  2. a scripting engine that reacts on this behavior
  3. a mechanism for asynchronous queries

## **Dynamic webpages**

---

# The not so simple world of dynamic webpages

## Dynamic webpages

- with dynamic webpages, the displayed content can differ between users, even if the source code is the same
- things that can cause the display of different content:
  - operating system, browser, device
  - user actions on the page (mouse movements, scrolling, clicks, keyboard strokes)
  - conditions on the server-side



Source: <https://xkcd.com/869/> (Randall Munroe)

# The not so simple world of dynamic webpages

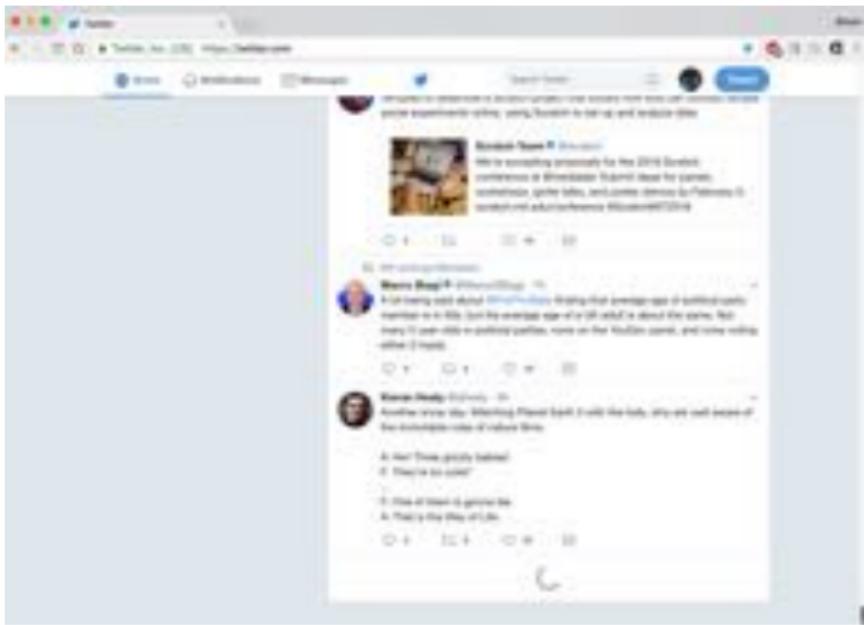
## Dynamic webpages

- massively used in modern webpage design and architecture
- (are thought to) enhance the user experience
- allow for much more ways to interact with webpage content

## Technologies

- in the case of **server-side** dynamic webpages, scripts on the server control webpage construction (e.g., PHP script reacts to user input to a form and returns subsets of a database)
- in the case of **client-side** dynamic webpages, (typically) JavaScript embedded in HTML determine what is displayed and how the DOM is changed
- a combination of these technologies is **Asynchronous JavaScript and XML (AJAX)**

# Examples of dynamic webpages



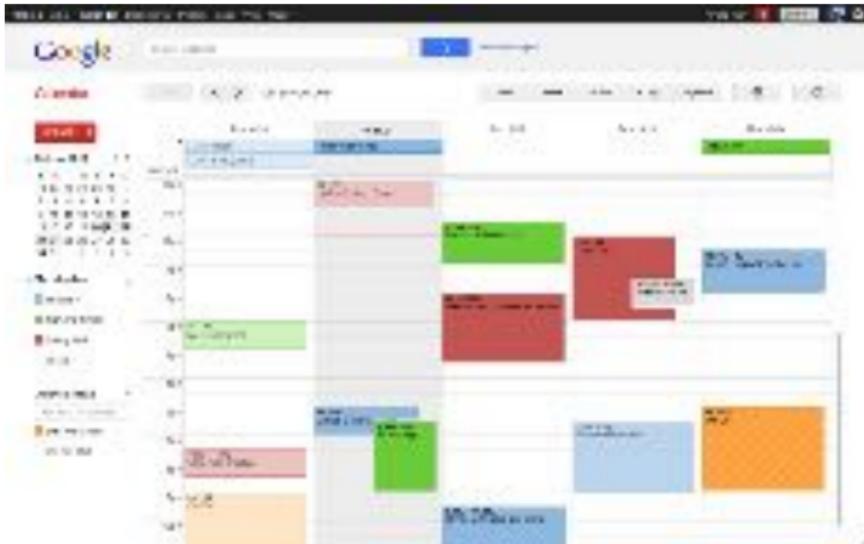
Your Twitter or Facebook feed that automatically gets updated when you scroll down

## Examples of dynamic webpages



The map application that automatically loads new content when you zoom in

## Examples of dynamic webpages



The calendar app that displays personal information and lets you interact a lot

## **The problem of dynamic webpages**

---

# The problem of dynamic webpages

- the tools you have encountered so far operated on the static source code: you access a page, download it, parse it
- but what if content on the page changes, but the source code does not?
- dynamic webpages make classical screen scraping more difficult if not impossible
- and: dynamically rendered webpages become more and more common
- before we learn about tools that can help us extract data in such scenarios, we will briefly consider the underlying processes, in particular AJAX technologies

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

AJAX Technologies

---

Simon Munzert

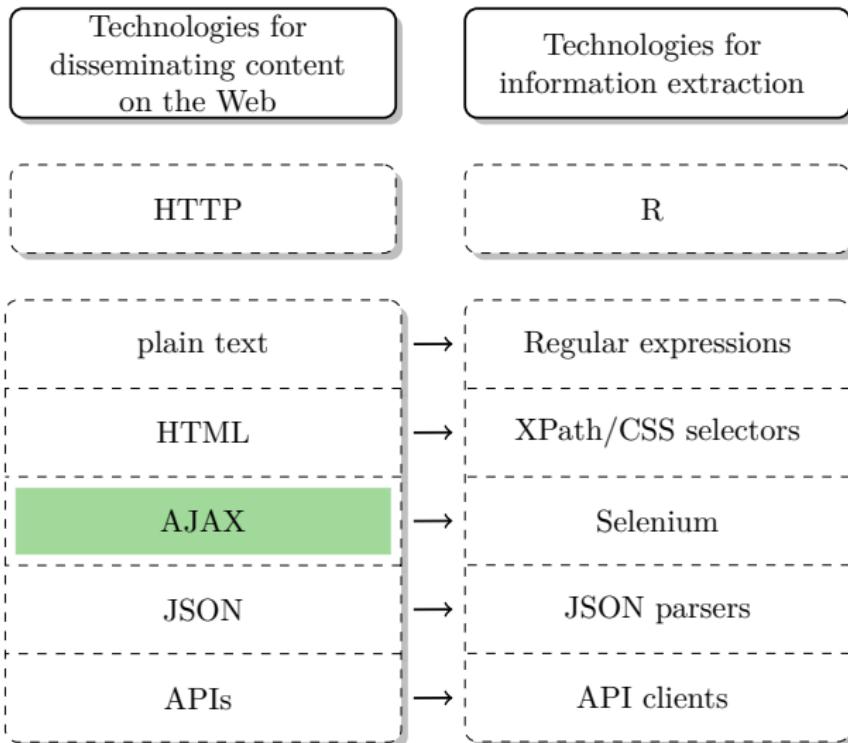
Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# AJAX

---

# Technologies of the World Wide Web



# What's AJAX?

## Purpose

- recall: HTML/HTTP lack
  1. mechanisms to detect user behavior in the browser
  2. a scripting engine that reacts on this behavior
  3. a mechanism for asynchronous queries
- **A**synchronous **J**ava**S**cript **a**nd **X**ML is a set of technologies that serve these purposes

## Components

- HTML/CSS for presentation
- Document Object Model (DOM; “tree structure”) to interact with data
- JSON/XML for data interchange
- XMLHttpRequest for asynchronous communication
- JavaScript as a scripting language

# JavaScript

---

## What's JavaScript? I

- Programming language that connects well to web technologies
- W3C web standard
- native browser support (built-in JS engine)
- nowadays employed on the majority of websites
- extensible by libraries, e.g. *jQuery*



## What's JavaScript? II

- technology to make webpages interactive ('dynamic HTML')
- allows to...
  - animate page elements
  - offer interactive content (games, video, ...)
  - manipulate page content and communication with the server without reloading the page

## How's JavaScript code embedded in HTML?

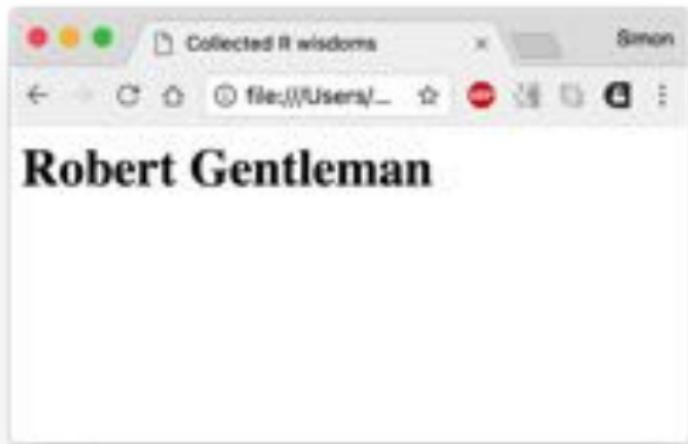
- between `<script>` tags
- as an external reference in the `src` attribute of a `<script>` element
- directly in certain HTML attributes ('event handler')

## DOM manipulation with JavaScript

- adding/removing HTML elements
- changing attributes
- modification of CSS styles
- ...

Example:

```
1 <script type="text/javascript" src="jquery-1.8.0.min.js"></script>
2 <script type="text/javascript" src="script1.js"></script>
```



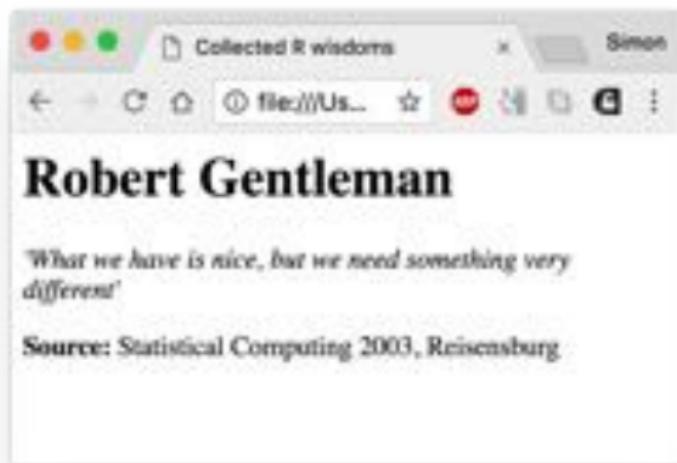
# JavaScript on the Web

```
1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
2 <html>
3
4 <script type="text/javascript" src="jquery-1.8.0.min.js"></script>
5 <script type="text/javascript" src="script1.js"></script>
6
7 <head>
8 <title>Collected R wisdoms</title>
9 </head>
10
11 <body>
12 <div id="R Inventor" lang="english" date="June/2003">
13   <h1>Robert Gentleman</h1>
14   <p><i>'What we have is nice, but we need something very different'</i>
15     </p>
16   <p><b>Source: </b>Statistical Computing 2003, Reisenburg</p>
17 </div>
18 </body>
19 </html>
```

## A JavaScript code snippet

```
1 $(document).ready(function() {  
2     $("p").hide();  
3     $("h1").click(function(){  
4         $(this).nextAll().slideToggle(300);  
5     });  
6 });
```

- `$()` operator: addresses DOM elements
- `ready()`: JavaScript execution starts when the complete DOM is ready, i.e. fetched from the server
- `hide()`: element is hidden at first place
- `click` event: identifies mouse click and executes a certain action
- `nextAll()`: all subsequent elements in the DOM are addressed
- `slideToggle()`: Toggle effect, 300 milli-seconds



## **Summary**

---

# Summary

- AJAX technologies allow for a dynamic manipulation of the HTML tree
- what the user sees in the browser is not necessarily what's in the static HTML source code
- elements can be added, removed, or changed
- the "live" DOM tree that you saw in the Web Developer Tools takes track of such changes
- but you cannot simply access this live HTML with R and **rvest**
- we need another technology that helps us mimic a session in the browser to render all dynamic content



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Selenium: Basics

---

Simon Munzert

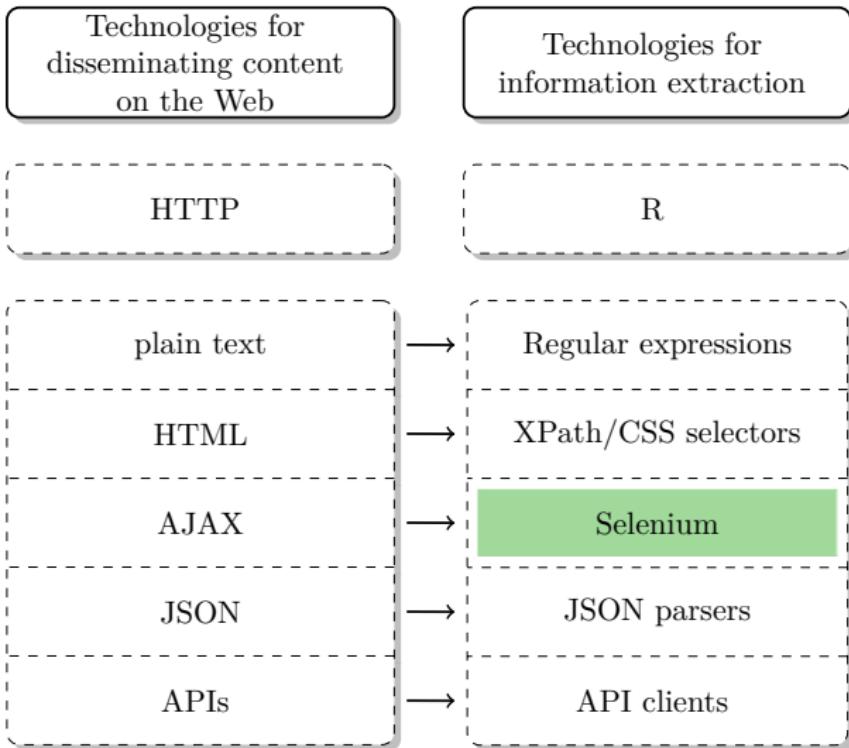
Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Selenium

---

# Technologies of the World Wide Web



## The problem reconsidered

- dynamic data requests are not stored in the static HTML page
- therefore, we cannot access them with classical methods and packages (`httr`, `rvest`, `download.file()`, etc.)
- R is not a browser with a JavaScript rendering engine

## The solution

- initiate and control a web browser session with R
- let the browser do the JavaScript interpretation work and the manipulations in the live DOM tree
- access information from the web browser session

## What's Selenium?

- free software environment for automated web application testing
- webpage: <http://www.seleniumhq.org>
- several modules for different tasks; most important for our purposes: Selenium WebDriver
- enables automated browsing via scripts



# Selenium and R

---

# The scraping workflow with Selenium and R

1. Selenium—an external, Java-based program—is launched
2. via the R package **RSelenium**, we remote-control Selenium and let it start a virtual server
3. we let Selenium start a browser session (e.g., Chrome)
4. everything we do in the browser—open a page, click on links or buttons, enter information—is described in R and sent to the server via the “remote-control” Selenium
5. we can gather the live HTML tree at any instance



## Software requirements

- Java, <https://www.java.com/de/download/>
- Selenium Standalone Server, newest version available here:  
<http://www.seleniumhq.org/download/> or via RSelenium and  
`rsDriver()`
- browser (on most systems, both Chrome and Firefox seem to work reliably)
- **RSelenium** package

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Selenium: Case Study

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **Selenium: Case Study**

---

# Example

## Overview

- **goal:** scrape data from the IEA/IRENA database
- URL:  
[http://www.iea.org/  
policiesandmeasures/  
renewableenergy/](http://www.iea.org/policiesandmeasures/renewableenergy/)
- the query form has multiple parameters
- the output comes in form of an HTML table, but content is injected into DOM



# Example

## Overview

- **goal:** scrape data from the IEA/IRENA database
- URL:  
<http://www.iea.org/policiesandmeasures/renewableenergy/>
- the query form has multiple parameters
- the output comes in form of an HTML table, but content is injected into DOM



## Example

## Overview

- **goal:** scrape data from the IEA/IRENA database
  - URL:  
<http://www.iea.org/policiesandmeasures/renewableenergy/>
  - the query form has multiple parameters
  - the output comes in form of an HTML table, but content is injected into DOM

# Example

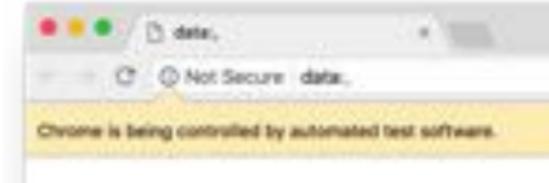
## Setup

- load **RSelenium**
- check installed Java version (helpful for debugging if Selenium fails to launch)
- initiate SeleniumDriver and browser

R code —————

```
1 library(RSelenium)
2 system("java -version")
  java version "9"
  Java(TM) SE Runtime
  Environment (build 9+181)
  Java HotSpot(TM) 64-Bit
  Server VM (build 9+181,
  mixed mode)
3 rD <- rsDriver()
4 remDr <- rD[["client"]]
```

————— end



# Example

## R calls

- `navigate()` to URL
- page opens  
"automatically"

R code —————

```
5 url <- "http://www.iea.org/  
policiesandmeasures/  
renewableenergy/"  
6 remDr$navigate(url)
```

————— end



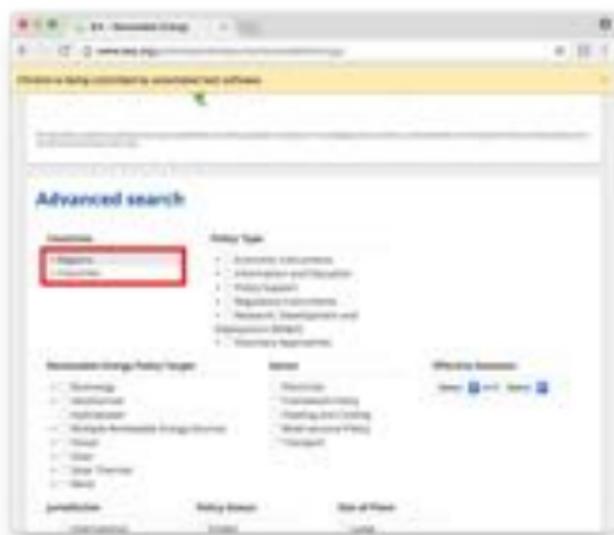
# Example

## R calls

- click on the "Regions" menu to unfold it
- extract XPath expression from Web Developer Tools
- pass XPath to `findElement()`, then click on it with `clickElement()`

R code —

```
xpath <- '//*[@id="main"]/div/form/div  
[1]/ul/li[1]/span'  
regionsElem <- remDr$findElement(using  
= 'xpath', value = xpath)  
openRegions <- regionsElem$clickElement  
()  
————— end
```



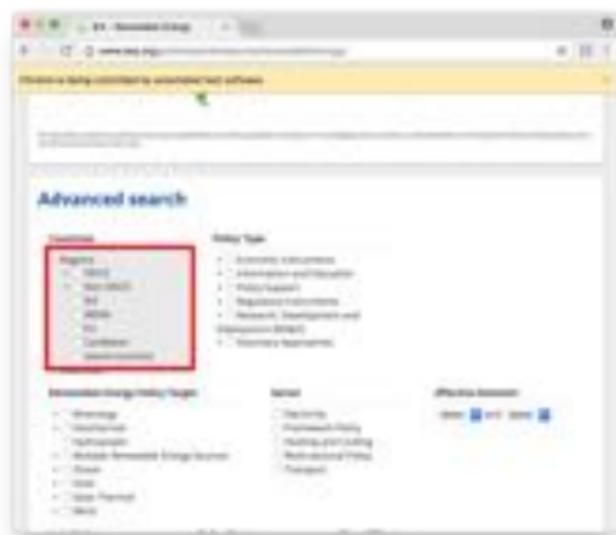
# Example

## R calls

- click on the "Regions" menu to unfold it
- extract XPath expression from Web Developer Tools
- pass XPath to `findElement()`, then click on it with `clickElement()`

R code —

```
xpath <- '//*[@id="main"]/div/form/div  
[1]/ul/li[1]/span'  
regionsElem <- remDr$findElement(using  
= 'xpath', value = xpath)  
openRegions <- regionsElem$clickElement  
()  
————— end
```



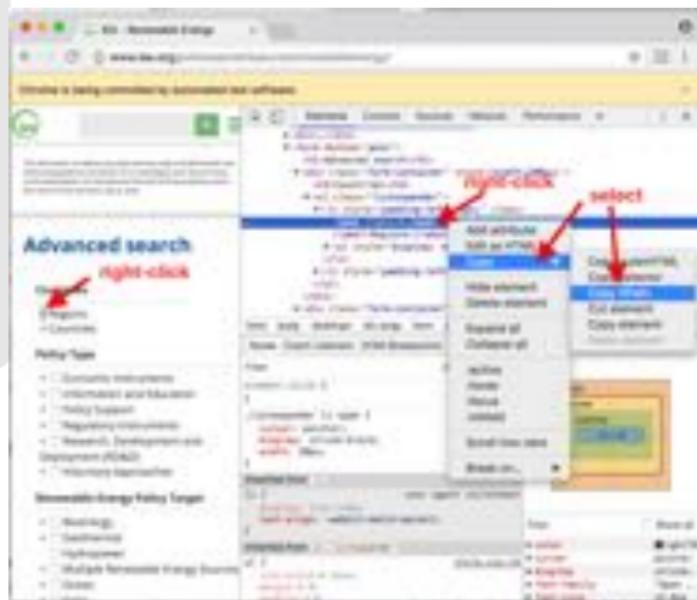
# Example

## R calls

- click on the "Regions" menu to unfold it
- extract XPath expression from Web Developer Tools
- pass XPath to `findElement()`, then click on it with `clickElement()`

R code

```
xpath <- '//*[@id="main"]/div/form/div[1]/ul/li[1]/span'  
regionsElem <- remDr$findElement(using =  
= 'xpath', value = xpath)  
openRegions <- regionsElem$clickElement  
()  
  
end
```



## Example

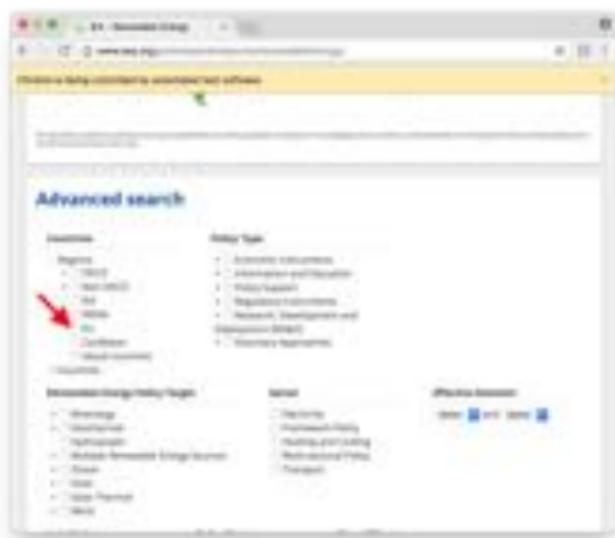
R calls

- select "EU" option
  - extract XPath expression from Web Developer Tools
  - pass XPath to `findElement()`, then click on it with `clickElement()`

R code

```
xpath <- '//*[@id="main"]/div/form/div[1]/ul/li[1]/ul/li[5]/label/input'  
euElem <- remDr$findElement(using = 'xpath', value = xpath)  
selectEU <- euElem$clickElement()
```

- end



# Example

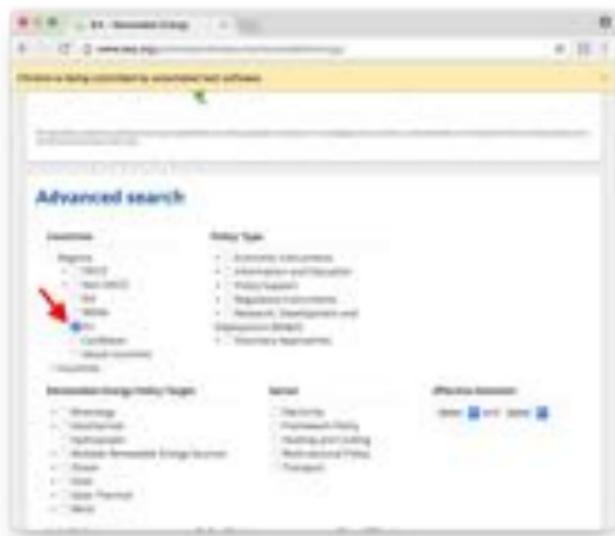
## R calls

- select "EU" option
- extract XPath expression from Web Developer Tools
- pass XPath to `findElement()`, then click on it with `clickElement()`

R code

```
xpath <- '//*[@id="main"]/div/form/div[1]/ul/li[1]/ul/li[5]/label/input'  
euElem <- remDr$findElement(using = 'xpath', value = xpath)  
selectEU <- euElem$clickElement()
```

end



# Example

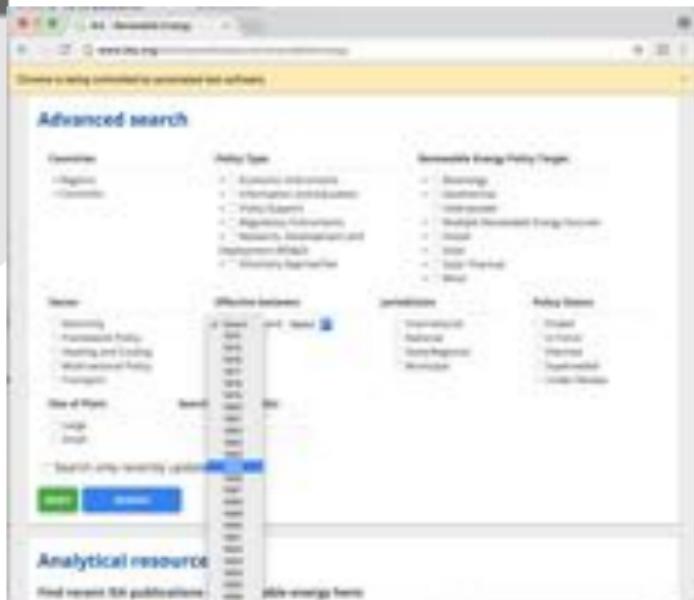
## R calls

- set time frame
- pass XPath to `findElement()`,  
then click on it, then enter text  
with `sendKeysToElement()`

R code —

```
xpath <- '//*[@id="main"]/div/form/div[5]/select[1]'  
fromDrop <- remDr$findElement(using = 'xpath', value = xpath)  
clickFrom <- fromDrop$clickElement()  
writeFrom <- fromDrop$sendKeysToElement(list("2000"))
```

— end



# Example

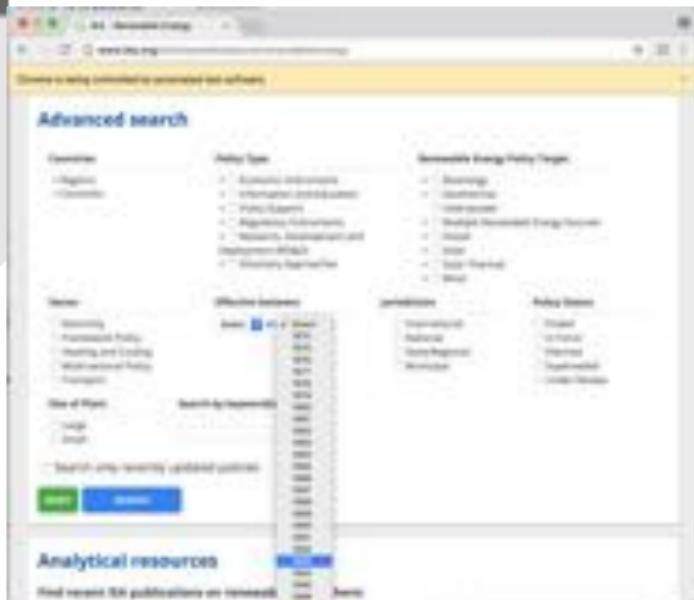
## R calls

- set time frame
- pass XPath to `findElement()`,  
then click on it, then enter text  
with `sendKeysToElement()`

R code —

```
xpath <- '//*[@id="main"]/div/form/div[5]/select[2]'  
fromDrop <- remDr$findElement(using = 'xpath', value = xpath)  
clickFrom <- fromDrop$clickElement()  
writeFrom <- fromDrop$sendKeysToElement(list("2010"))
```

— end



# Example

## R calls

- click on search button
- extract XPath expression of element from Web Developer Tools
- pass XPath to `findElement()`, then click on it

R code —————

```
xpath <- '//*[@id="main"]/div/form/  
button[2]'  
  
searchElem <- remDr$findElement(using =  
  'xpath', value = xpath)  
  
resultsPage <- searchElem$clickElement  

```

————— end



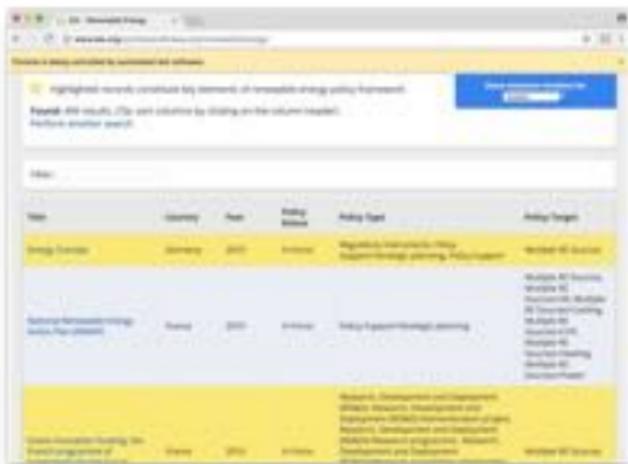
# Example

## R calls

- finally, we're there!
- now take snapshot of the DOM with `getPageSource()` and store it as HTML with `write()`

R code —

```
output <- remDr$getPageSource(header =  
TRUE)  
write(output[[1]], file = "iea-  
renewables.html")  
end
```



# Example

## Parsing HTML into data frame

- close the connection to Selenium server with `closeServer()`
- proceed with business as usual (`rvest` package)

R code

---

```
35 remDr$closeServer()
36 content <- read_html("iea-renewables.html", encoding = "utf8")
37 tabs <- html_table(content, fill = TRUE)
38 tab <- tabs[[1]]
39 "target")
40 tab[1,]

          title country year    status
                           type
1 Energy Concept Germany 2010 In Force Regulatory Instruments, Policy
Support>Strategic planning, Policy Support
          target
1 Multiple RE Sources
```

## **Summary**

---

# Summary

- Selenium is an excellent tool to scrape data from dynamic, JavaScript-enriched webpages where ordinary scraping methods fail
- once the setup is established (which can be troublesome, as many software components have to work together), its use is pretty simple
- I do not recommend it as a substitute for every scraping task because it is too slow and unreliable for that purpose



By W. Oelen (CC BY-SA 3.0),  
<https://commons.wikimedia.org/w/index.php?curid=15369617>

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

## Legal Issues

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# **Is web scraping legal?**

---

# Is web scraping legal?

## Disclaimer

Obviously, I am not a lawyer. Do not rely on any of my comments on this topic. If you are seriously worried about the legality of your scraping work, please consult a legal expert.



# Is web scraping legal?

## Scraping vs. crawling

- **Web scraping:** downloading data from a very specific page in a (semi-)automated manner
- **Web crawling:** automatically downloading webpage data, extracting hyperlinks, following links, downloading webpage data, ... (e.g.: Googlebot, BaiduSpider)

**This course mostly is scraping, not crawling or web harvesting!**

- web scraping per se is not illegal
- there is no unambiguous **yes** or **no** for specific applications in any country according to current jurisdiction
- so far, legal cases (especially in the US) often (but not always) dealt with commercial interest, crawling applications, and often (but not always) huge masses of data, e.g., *eBay vs. Bidder's Edge*, *AP vs. Meltwater*, *Facebook vs. Pete Warden*

# Is web scraping legal?

## Why web scraping can be problematic

- violation of copyrights, Terms of Service, consumption of bandwidth

## Some counter-arguments

- data is publicly accessible
  - but the website as a "creative arrangement" might be copyrighted
- this is fair use → depends on your use
- it's the same what my browser does
  - not exactly, and many ToS prohibit automated uses of their data
- this is unfair—Google's business model is built on crawling the whole web
  - true, but you are not Google

## Some interesting comments by Pablo Hoffman, co-founder of Scrapinghub

- As long as they don't crawl at a disruptive rate, scrapers do not breach any contract (in the form of terms of use) or commit a crime (as defined in the Computer Fraud and Abuse Act).
- Website's user agreement is not enforceable as a browse-wrap agreement because companies do not provide sufficient notice of the terms to site visitors.
- Scrapers accesses website data as a visitor, and by following paths similar to a search engine. This can be done without registering as a user (and explicitly accepting any terms).

(found on <https://goo.gl/jTt4ER>)

## Things webpage administrators can do to prevent you from scraping massive amounts of data from their pages

- block your IP address
- identify your approximate geolocation from your IP address, then block
- move content exclusively to web services / APIs
- block bots with a particular user agent string (more on that later)
- challenge-response tests like **C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part (CAPTCHA)
- obfuscation of data
- frequent changes in HTML/CSS

## **Summary**

---

# Summary

- there is no unconditional "legal" or "illegal" status of web scraping
- your use of the data can violate the data owner's rights
- targeted scraping efforts with limited traffic usually do not cause any problems



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Good Practice

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **Understanding robots.txt**

---

## What's robots.txt?

- ‘Robots Exclusion Protocol’, informal protocol to prohibit web robots from crawling content
- located in the root directory of a website (e.g.,  
<http://www.google.com/robots.txt>)
- documents which bot is allowed to crawl which resources (and which not)
- not a technical barrier, but a sign that asks for compliance

## Example: NY Times's robots.txt



# Syntax in robots.txt

## Syntax

- not an official W3C standard, partly inconsistent syntax
- rules listed bot by bot
- general, bot-independent rules under '\*' (most interesting entry for R-based crawlers)
- directories/folders listed separately

```
1 User-agent: Googlebot
2 Disallow: /images/
3 Disallow: /private/
```

```
1 User-agent: *
2 Disallow: /private/
```

# Syntax in robots.txt

## Universal ban

```
1 User-agent: *
2 Disallow: /
```

## Separation of bots by empty line

```
1 User-agent: Googlebot
2 Disallow: /images/
4 User-agent: Slurp
5 Disallow: /images/
```

## Allow declaration

```
1 User-agent: *
2 Disallow: /images/
3 Allow: /images/public/
```

# Syntax in robots.txt

## Crawl-delay (in seconds)

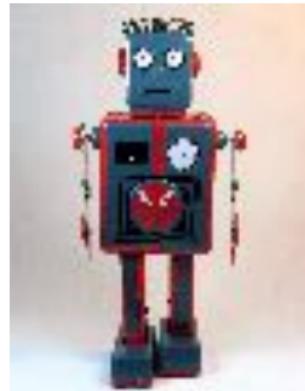
```
1 User-agent: *
2 Crawl-delay: 2
3 User-Agent: Googlebot
4 Disallow: /search/
```

## Robots <meta> tag

```
1 <meta name="robots" content="noindex,nofollow" />
```

# How to deal with robots.txt?

- not clear if robots.txt is legally binding or not, and if yes for which activities
- originally not thought of as protection against small-scale web scraping applications, but against large-scale indexing bots
- guide to a webmaster's preferences with regards to visibility of content
- my advice: take `robots.txt` into account! If the data you are interested in are excluded from crawling: contact webmaster
- there is even an R package, `robotstxt`, that helps you parse `robots.txt` documents and stick to the rules. It's available here:  
<https://github.com/ropenscilabs/robotstxt>



By D J Shin - My Toy Museum, [https://commons.wikimedia.org/wiki/File:QSH\\_Tin\\_Wind\\_Up\\_Mechanical\\_Robot\\_\(Giant\\_Easelback\\_Robot\)\\_Front.jpg](https://commons.wikimedia.org/wiki/File:QSH_Tin_Wind_Up_Mechanical_Robot_(Giant_Easelback_Robot)_Front.jpg)

## **Scraping etiquette**

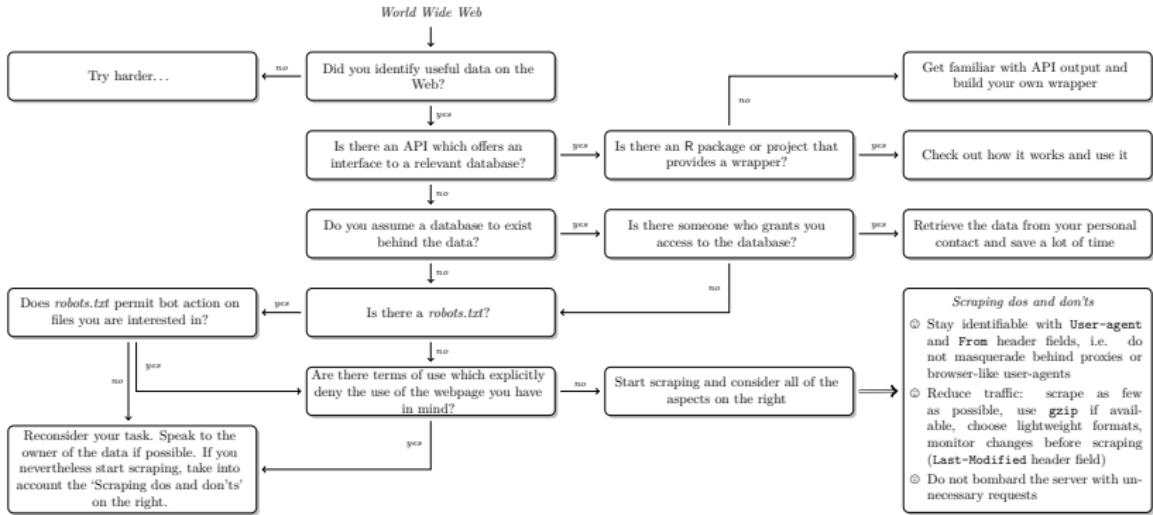
---

# Some advice for your work

---

1. You take all the responsibility for your web scraping work.
2. Take all copyrights of a country's jurisdiction into account.
3. If you publish data, do not commit copyright fraud.
4. If possible, stay identifiable.
5. If in doubt, ask the author/creator/provider of data for permission—if your interest is entirely scientific, chances aren't bad that you get data.
6. Consult current jurisdiction, e.g. on  
<http://blawgsearch.justia.com> or from a lawyer specialized on internet law.

# Scraping etiquette



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

APIs

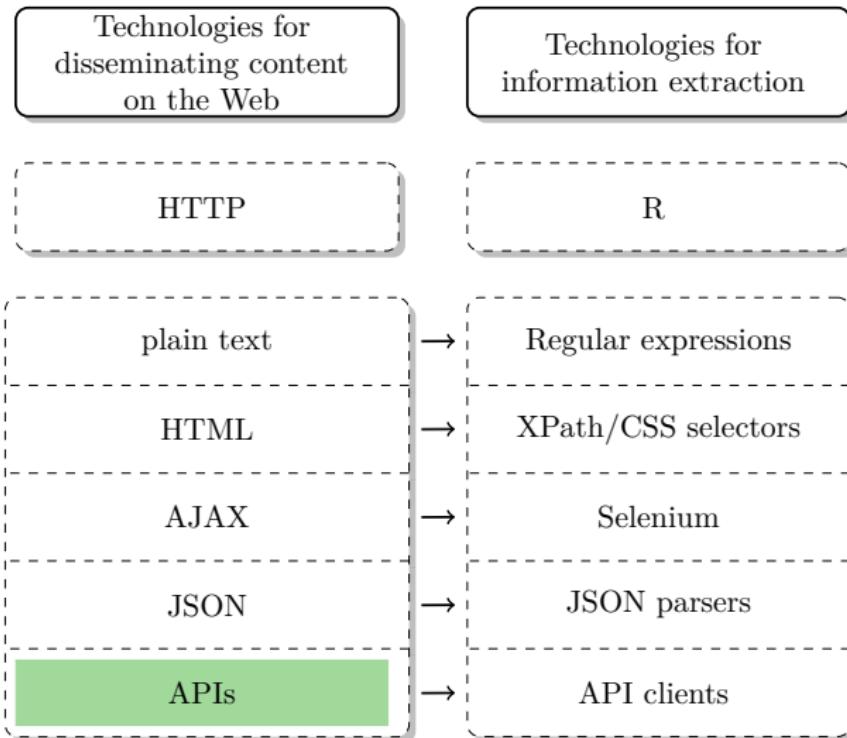
---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Technologies of the World Wide Web



## **What are APIs?**

---

# What are APIs?

## Definition

- Application Programming Interface
- "data search engine": you pose a request, the API answers with a bulk of data
- let you/your program query a provider for specific data
- common data formats: XML, JSON
- many popular web services provide APIs (Twitter, Google, Facebook, Wikipedia, ...)

## Why we should care about APIs

- provide instant access to clean data
- free us from building manual scrapers
- API usage implies mutual data collection agreement

## **Example**

---

# Example

## Google Maps API

- Google provides access to powerful location services
- free service (at least when used modestly)
- input/output: places, names, coordinates, maps, ...
- see also: <https://developers.google.com/maps/documentation/>



Google Maps

## Potential use cases

- geocode observations based on address, city, post code, ...
- calculate distances between observations
- map observations

# Example

## Access the API with R

- the **ggmap** package provides high-level functions to access API
- in this case, all we have to do is to figure out how the R function works – the communication with the API is processed completely in the background

R code

```
1 library(ggmap)
2 geocode("Berlin, Germany")
```

```
Information from URL : http://maps.googleapis.com/maps/api/geocode/json?
address=Berlin,%20Germany&sensor=false
      lon      lat
1 13.40495 52.52001
```

end

# Example

## Excerpt from raw JSON behind the call

```
1  {
2      "results" : [
3          {
4              "address_components" : [
5                  {
6                      "long_name" : "Berlin",
7                      "short_name" : "Berlin",
8                      "types" : [ "locality", "political" ]
9                  },
10                 ],
11                 "formatted_address" : "Berlin, Germany",
12                 "location" : {
13                     "lat" : 52.52000659999999,
14                     "lng" : 13.404954
15                 }
16             },
17             "place_id" : "ChIJAVkDPzd0qEcRcDteWOYgIQQQ",
18             "types" : [ "locality", "political" ]
19         }
20     ]
21 }
```

# Example

## Map the location

R code

```
3 get_googlemap("Berlin, Germany",
  zoom = 12, maptype = "hybrid")
%>% ggmap()
----- end
```



## **Summary**

---

# Summary

## Advantages of API use

- collecting data from the web using APIs provided by the data owner represents **the gold standard of web data retrieval**
- pure data collection without ‘layout waste’
- standardized data access
- de facto automatic agreement
- robustness of calls



[http://maxpixel.  
freegreatpicture.com/  
photo-1461569](http://maxpixel.freegreatpicture.com/photo-1461569)

## Disadvantages of API use

- (sometimes) requires knowledge of API architecture
- dependent upon API suppliers
- use not always free

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

API Clients

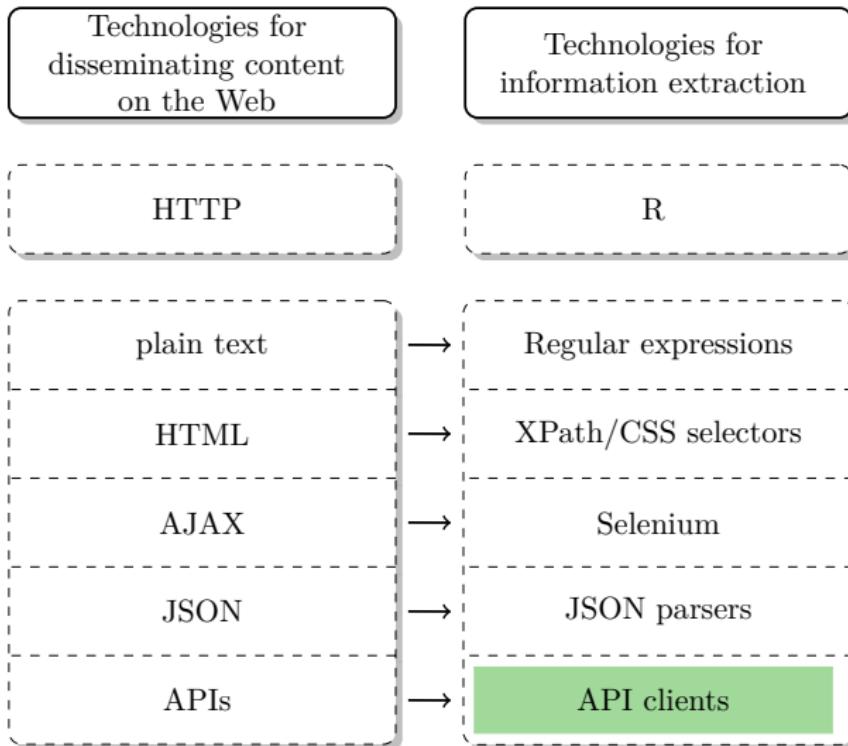
---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Technologies of the World Wide Web

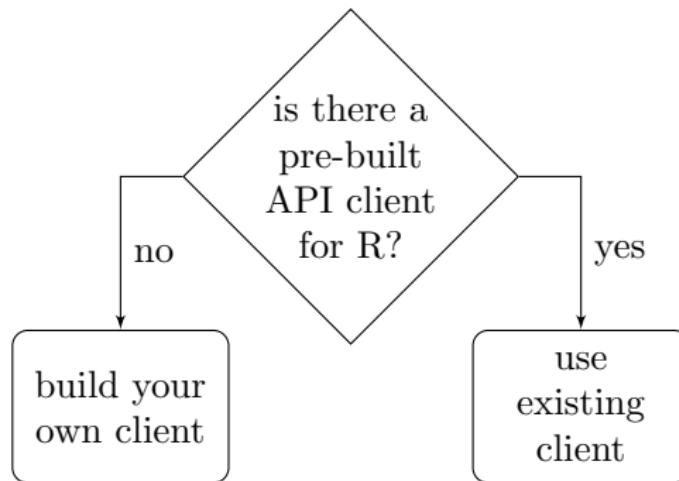


# **Accessing APIs with R**

---

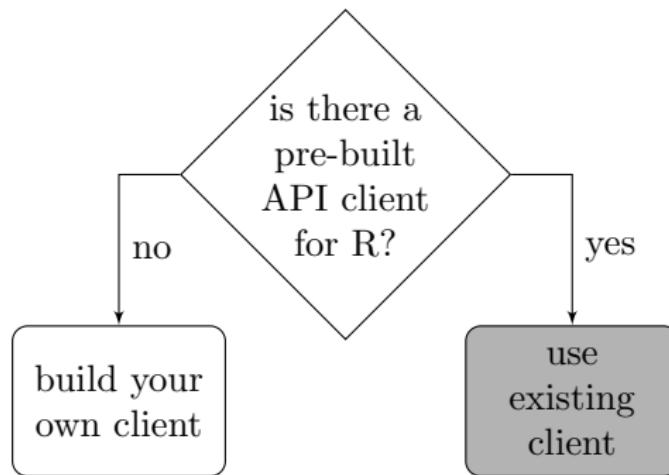
# API access with R

There are basically **two scenarios:**



# API access with R

There are basically **two scenarios:**



Here, we talk about the case where a client already exists.

## API clients

- provide interface to APIs
- hide API back-end
- let you stay in your programming environment

## Example

- the `rtweet` package provides an R client for Twitter
- lets you query data from the Twitter API with R commands
- you don't have to work with unfamiliar data formats (JSON) that is provided by the API—the client automatically transforms the incoming data into R objects

# Finding API clients on the Web

## General resources

List of APIs: <http://www.programmableweb.com/apis>

rOpenSci – collection of R API clients:

<https://github.com/ropensci/opendata>

CRAN Task View:

<http://cran.r-project.org/web/views/WebTechnologies.html>

## How to find the API client you need

- google "R package + name of website"
- search on the website for a "Developer" or "API" section (only if you don't find an R package that works)

# Popular R API clients

package name	access to	more info
<code>rtweet</code>	Twitter Stream and REST API	<a href="http://rtweet.info/">http://rtweet.info/</a>
<code>Rfacebook</code>	Facebook API	<a href="https://github.com/pablobarbera/Rfacebook">https://github.com/pablobarbera/Rfacebook</a>
<code>ipapi</code>	ip-api.com's API	<a href="https://github.com/hrbrmstr/ipapi">https://github.com/hrbrmstr/ipapi</a>
<code>ggmap</code>	Google Maps/OpenStreetMap APIs	<a href="https://github.com/dkahle/ggmap">https://github.com/dkahle/ggmap</a>
<code>eurostat</code>	Eurostat database	<a href="https://github.com/ropengov/eurostat">https://github.com/ropengov/eurostat</a>
<code>rftimes</code>	New York Times APIs	<a href="https://cran.rstudio.com/web/packages/rftimes/index.html">https://cran.rstudio.com/web/packages/rftimes/index.html</a>

## **Summary**

---

# Summary

- if the website you want to use as a data basis for your research provides access via an API, you can consider yourself lucky
- if there is a working R-based API client available, you hit the jackpot



## Final considerations

- please always cite package authors when you use their work. run `citation("<package name>")` to see how
- sometimes API clients are not up to date. Consider to notify the author or help update the package
- sometimes API clients provide functionality for only a fraction of the API's capability. It's always worth to check out the API documentation if you are looking for specific features

Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

JSON

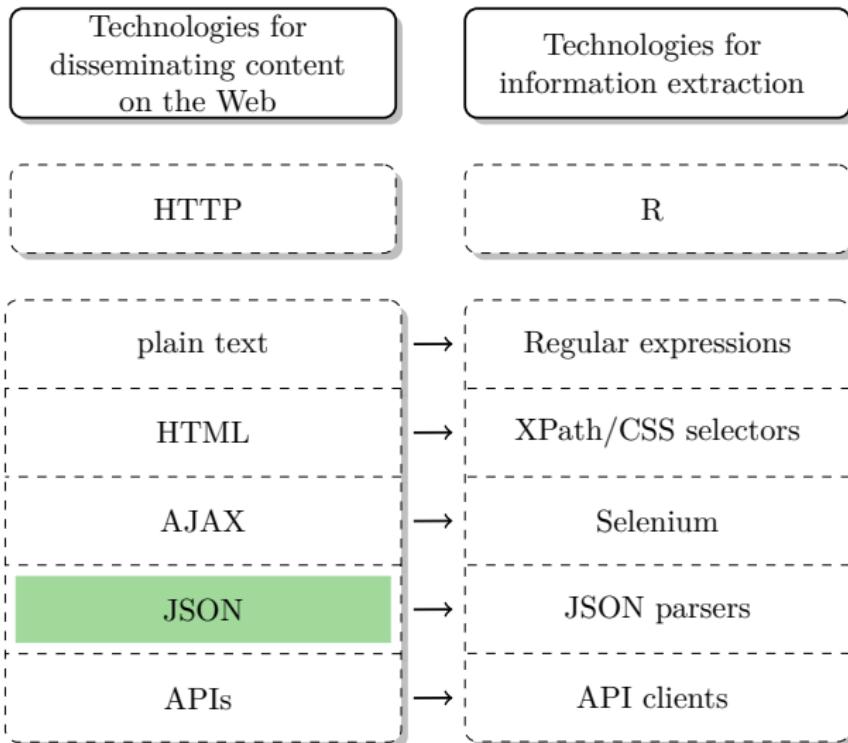
---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

# Technologies of the World Wide Web



- **JavaScript Object Notation**
- popular data exchange format for web services / APIs
- ‘the fat-free alternative to XML’
- JSON ≠ Java, but a subset of JavaScript
- however, very flexible and not dependent upon any programming language
- import of JSON data into R is relatively straightforward with the **jsonlite** package



# Example

```
1 {"indy movies": [
2     {"name": "Raiders of the Lost Ark",
3      "year": 1981,
4      "actors": {
5          "Indiana Jones": "Harrison Ford",
6          "Dr. Rene Belloq": "Paul Freeman"
7      },
8      "producers": ["Frank Marshall", "George Lucas", "Howard Kazanjian"],
9      "budget": 18000000,
10     "academy_award_ve": true},
11     {"name": "Indiana Jones and the Temple of Doom",
12      "year": 1984,
13      "actors": {
14          "Indiana Jones": "Harrison Ford",
15          "Mola Ram": "Amish Puri"
16      },
17      "producers": ["Robert Watts"],
18      "budget": 28170000,
19      "academy_award_ve": true}
20 ]
21 }
```

## Types of brackets

1. curly brackets, ‘{’ and ‘}’, embrace **objects**. Objects work similar to elements in XML/HTML and can contain other objects, key-value pairs or arrays
2. square brackets, ‘[’ and ‘]’, embrace **arrays**. An array is an ordered sequence of objects or values.

# Key-value pairs

1. Keys are put in quotation marks; values only if they contain string data.

```
1 "name" : "Indiana Jones and the Temple of Doom"  
2 "year" : 1984
```

2. Keys and values are separated by a colon.

```
1 "year" : 1981
```

3. Key-value pairs are separated by commas.

```
1 {"Indiana Jones": "Harrison Ford",  
2 "Dr. Rene Belloq": "Paul Freeman"}
```

4. Values within arrays are separated by commas.

```
1 ["Frank Marshall", "George Lucas", "Howard Kazanjian"]
```

# Data types

## Summary

- JSON allows a basic set of data types
- often equivalent data types available in different programming languages
- compatibility with R partly given, but no isomorphic translation possible

data type	meaning
number	integer, real, or floating point (e.g., 1.3E10)
string	whitespace, zero or more Unicode characters (except " or \; \ introduces some escape sequences)
boolean	<code>true</code> or <code>false</code>
null	<code>null</code> , an unknown value
Object	content in curly brackets
Array	ordered content in square brackets

## Parsing software

- different packages available for R: `rjson`, `RJSONIO`, `jsonlite`
- choose `jsonlite`: it's under active development and provides convincing mapping rules

# JSON and R

## Parsing JSON with `jsonlite`

R code

---

```
1 (indy <- fromJSON("../materials/indy.json"))
$`indy movies`  
name year actors.Indiana Jones  
1 Raiders of the Lost Ark 1981 Harrison Ford  
2 Indiana Jones and the Temple of Doom 1984 Harrison Ford  
3 Indiana Jones and the Last Crusade 1989 Harrison Ford  
actors.Dr. Rene Belloq actors.Mola Ram actors.Walter Donovan  
1 Paul Freeman <NA> <NA>  
2 <NA> Amish Puri <NA>  
3 <NA> <NA> Julian Glover  
producers budget academy_award_
ve  
1 Frank Marshall, George Lucas, Howard Kazanjian 18000000  
TRUE  
2 Robert Watts 28170000  
TRUE  
3 Robert Watts, George Lucas 48000000  
FALSE
```

## Mapping rules of jsonlite

R code

---

```
2 library(jsonlite)
3 x <- '[1, 2, true, false]'
4 fromJSON(x)
[1] 1 2 1 0
5 x <- '["foo", true, false]'
6 fromJSON(x)
[1] "foo"    "TRUE"   "FALSE"
7 x <- '[1, "foo", null, false]'
8 fromJSON(x)
[1] "1"      "foo"    NA       "FALSE"
```

---

end

- there is no ultimate JSON-to-R converting function
- **jsonlite** simplifies matters a lot
- usually, JSON files returned by web services are not too complex



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

Accessing APIs from Scratch

---

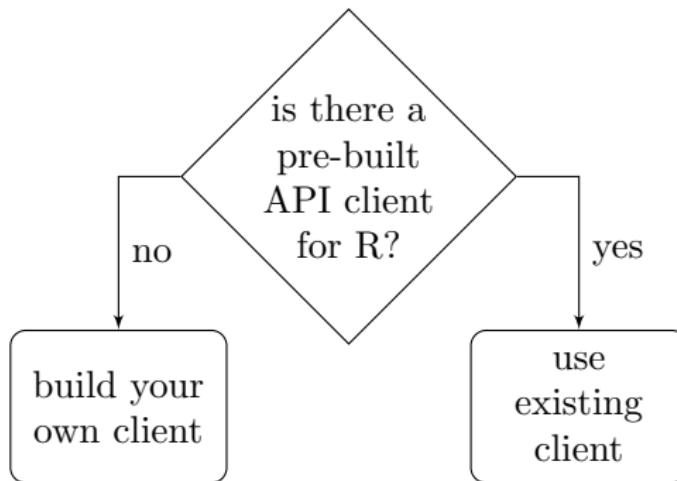
Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

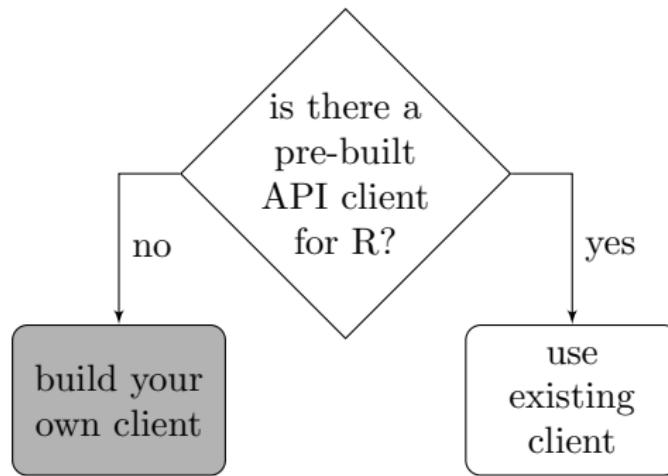
# API access with R

There are basically **two scenarios**:



# API access with R

There are basically **two scenarios:**



Here, we talk about the case where you have to build your own API binding.

# **Accessing APIs from Scratch**

---

## Steps to be taken

1. figure out how the API works: every API has a human-readable documentation for developers
2. build access to the API via R
3. build functionality that processes the data output (e.g., turns it into R objects)

## **Example**

---

## Example

# The IP API

- available at  
<http://ip-api.com/>
  - its purpose: parses your IP (or a bunch of given IPs) and returns some values, including guessed location, service provider, and organization behind the IP



# Example

## The IP API

- available at  
<http://ip-api.com/>
- its purpose: parses your IP (or a bunch of given IPs) and returns some values, including guessed location service provider, and organization behind the IP
- check out API documentation



# Example

## The IP API

- access is free for non-commercial use (up to 150 requests per minute)
- various response formats available
- paid pro service available



# Example

## The IP API

- to access the API, we have to send a **GET** request to <http://ip-api.com/json>
- we can provide any IP we want
- the response is raw JSON code



A screenshot of a web browser displaying the JSON response from the IP API. The URL in the address bar is `http://ip-api.com/json`. The page title is "IP API". On the left, there's a sidebar with a tree view of API endpoints. The main content area is titled "JSON" and contains two sections: "Usage" and "Response". The "Usage" section provides instructions for making requests. The "Response" section shows a block of JSON code. The JSON output includes fields like "status", "query", "country", "countryCode", "region", "regionName", "city", "lat", "lon", "zip", "org", "as", "isp", "proxy", "blocklisted", and "error". A red box highlights the "Response" section.

```
{"status":"success","query":"192.168.1.100","country":"United States","countryCode":"US","region":"New York City","regionName":"New York City","city":"New York","lat":"40.7128","lon":"-74.0060","zip":"10001-3700","org":"AS33562 AT&T U.S. IP Network","as":"AS33562 AT&T U.S. IP Network","isp":"AT&T U.S. IP Network","proxy":"false","blocklisted":"false","error":""}
```

# Example

## IP API access in R

A **GET** request essentially means that we assemble a URL using

- the API **endpoint** (a basic URL) and
- **parameters-value pairs** that are added to the URL
- here, we only add the literal IP address
- we use **fromJSON** to pose the request and directly parse the data from the API

R code

```
1 url <- "http://ip-api.com/json"
2 url_ip <- paste0(url, "/208.80.152.201")
3 ip_parsed <- jsonlite::fromJSON(url_ip)
```

end

# Example

## Investigating the output

R code

---

```
4 names(ip_parsed)
[1] "as"          "city"        "country"      "countryCode"  "isp"
[6] "lat"         "lon"         "org"         "query"       "region"
[11] "regionName" "status"      "timezone"    "zip"

5 ip_parsed
$as
[1] "AS14907 Wikimedia Foundation Inc."

$city
[1] "San Francisco"

$country
[1] "United States"

$countryCode
[1] "US"
```

# Example

## Bringing it into shape

- in our case, `fromJSON()` has created an R list object
- we turn it into a data frame

R code

---

```
6 ip_parsed %>% as.data.frame(stringsAsFactors = FALSE)
               as      city      country
1 AS14907 Wikimedia Foundation Inc. San Francisco United States
   countryCode          isp      lat      lon
1           US Wikimedia Foundation Inc. 37.787 -122.4
               org      query region regionName status
1 Wikimedia Foundation Inc. 208.80.152.201       CA California success
   timezone      zip
1 America/Los_Angeles 94105
```

---

end

## **Summary**

---

# Summary

- accessing APIs from scratch can be almost as easy as using a readily available R client
- often however, APIs are more complex (provide much more parameters and variations in data output)
- ideally, you build functions around your API calls to make it even more accessible



Max-Planck-Institut für Gesellschaftsforschung

# A Primer to Web Scraping with R

API Authentication

---

Simon Munzert

Hertie School of Governance, Berlin

Jan 31 - Feb 01, 2019

## **API access limits**

---

# Why API access can be restricted

- service provider wants to know who uses their API
- hosting APIs is costly—API usage limits can help control costs
- commercial interest of API hoster: you pay for access  
(sometimes for advanced features or massive amounts of queries only)

## Access tokens

- access tokens serve as the key to the API
- they usually come in form of a randomly generated string, such as `dk5nSj485jJZP3847kjU`
- obtaining a token requires registration (often email address is sufficient)
- sometimes you have to disclose your intentions
- once you have the token, you usually pass it along with your regular API query

## **Example**

---

# Example

## The OpenWeatherMap API

- available at <http://openweathermap.org>
- free access to basic weather data (paid plans for historical and more detailed data)
- sign-up necessary



# Example

## The OpenWeatherMap API

- available at <http://openweathermap.org>
- free access to basic weather data (paid plans for historical and more detailed data)
- sign-up necessary
- sign-up via browser—name and email address suffices



# Example

## Accessing the API from R

- copy the key to a string and store it in a local file
- import the key when you want to tap the API

R code —

---

```
1 openweathermap <- "k4875jHkdfd9kBbiuZ208d7s"  
2 save(openweathermap, file = "/Users/s.munzert/rkeys.RDa")
```

end

---

R code —

---

```
3 load("/Users/s.munzert/rkeys.RDa")  
  
Error in readChar(con, 5L, useBytes = TRUE): cannot open the connection  
4 apikey <- paste0("&appid=", openweathermap)  
  
Error in paste0("&appid=", openweathermap): object 'openweathermap' not  
found
```

end

---

# Example

## Accessing the API from R

- send along the API key when you make queries to the API

R code

```
5 endpoint <- "http://api.openweathermap.org/data/2.5/find?"  
6 city <- "Berlin, Germany"  
7 metric <- "&units=metric"  
8 url <- paste0(endpoint, "q=", city, metric, apikey)  
  
Error in paste0(endpoint, "q=", city, metric, apikey): object 'apikey'  
not found  
9 jsonlite::fromJSON(url, flatten = TRUE)$list[1, ]  
  
Error: Argument 'txt' must be a JSON string, URL or file.
```

end

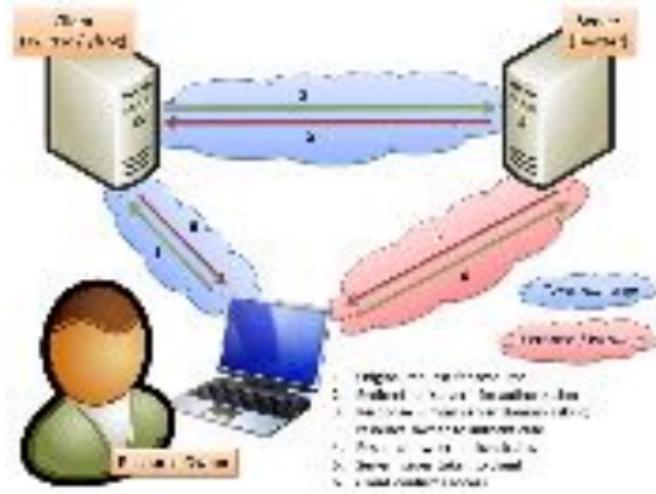
## **OAuth authorization**

---

# Accessing APIs with OAuth authorization

## What's OAuth?

- authorization standard
- used to provide client applications access to owner's resources—without giving them passwords
- frequently used by major companies (Twitter, Facebook, Amazon, Google, ...) to allow third party applications to interact with user's accounts



Source: <http://www.ubelly.com/wp-content/uploads/2010/02/OAuth1.png>

# Accessing APIs with OAuth authorization

## The OAuth workflow with `httr`

- `oauth_endpoint()`: define OAuth endpoints for the request and access token
- `oauth_app()`: bundle key and secret to request access credentials
- `oauth1.0_token()` and `oauth2.0_token()`: exchange consumer key and secret for access key and secret

... but often the R API client simplifies matters (e.g., the `rtweet` package)