A Primer to Web Scraping with R

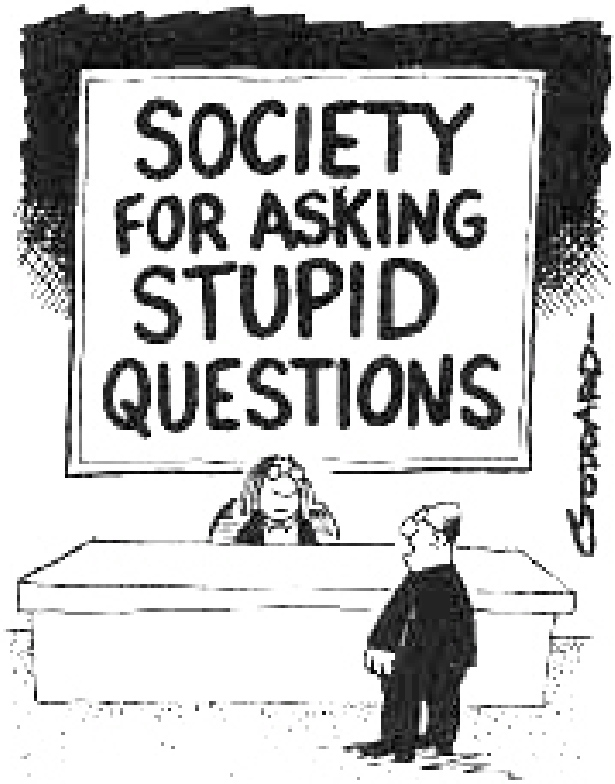
Simon Munzert, University of Konstanz
r-datacollection.com

March 2015

Workshop outline

Introduction
Regular Expressions
HTML and XPath
JSON
APIs
AJAX and Selenium
Advanced Scraping Applications
Good Practice

First: ask questions! No matter what...



"Excuse me, but is this The Society for Asking Stupid Questions?"

Part I

Introduction

Overview

Introduction

- Organizational Matters
- Why Web Scraping?
- Why R?
- Technologies of the World Wide Web
- Technical Setup
- A first encounter with the Web using R
- Exercises

Workshop outline

Time	Topic
26.03.2015, 09:00 -12:00	Introduction; a first encounter with the Web using R
26.03.2015, 13:15 -15:15	Scraping with regular expressions
26.03.2015, 15:45 -18:00	Scraping via XPath
27.03.2015, 09:00 -12:00	Social Media, APIs and JSON
27.03.2015, 13:15 -15:15	Scraping data from AJAX-enriched webpages
27.03.2015, 15:45 -18:00	Advances scraping scenarios and etiquette

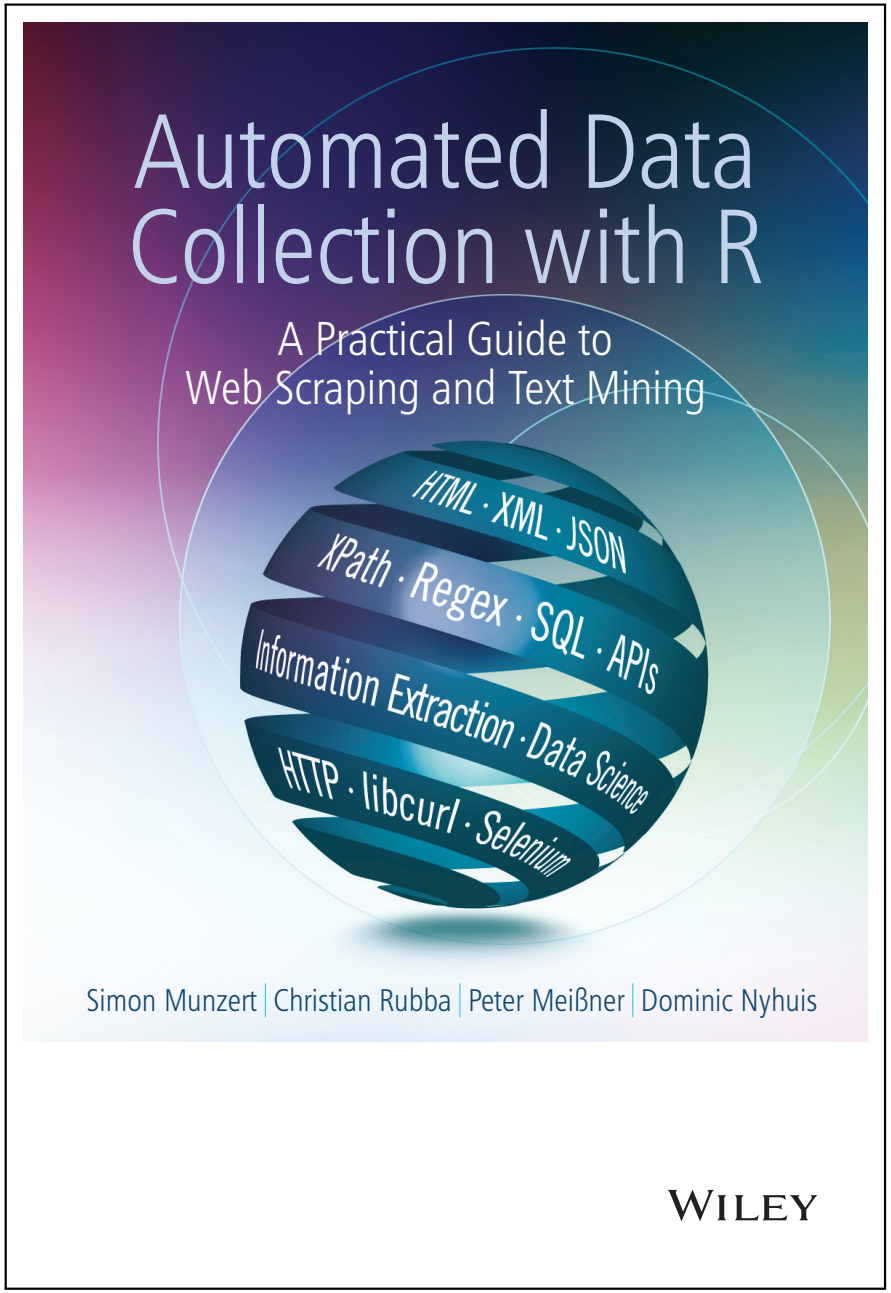
Goals

After attending this course, you ...

- have 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 data sets

The accompanying book

- contains most of which I tell you during the workshop (but much more, and much more accurate)
- written between 2012 and 2014 → not entirely up-to-date anymore, more on that later
- homepage with materials: www.r-datacollection.com
- use the manuscript, but don't give it away
- if you find any errors in the book, please tell us!



Web scraping. What? Why?

Web scraping

A.k.a. screen scraping, crawling, web harvesting; computer-aided collection of predominantly unstructured data (e.g., from HTML code)

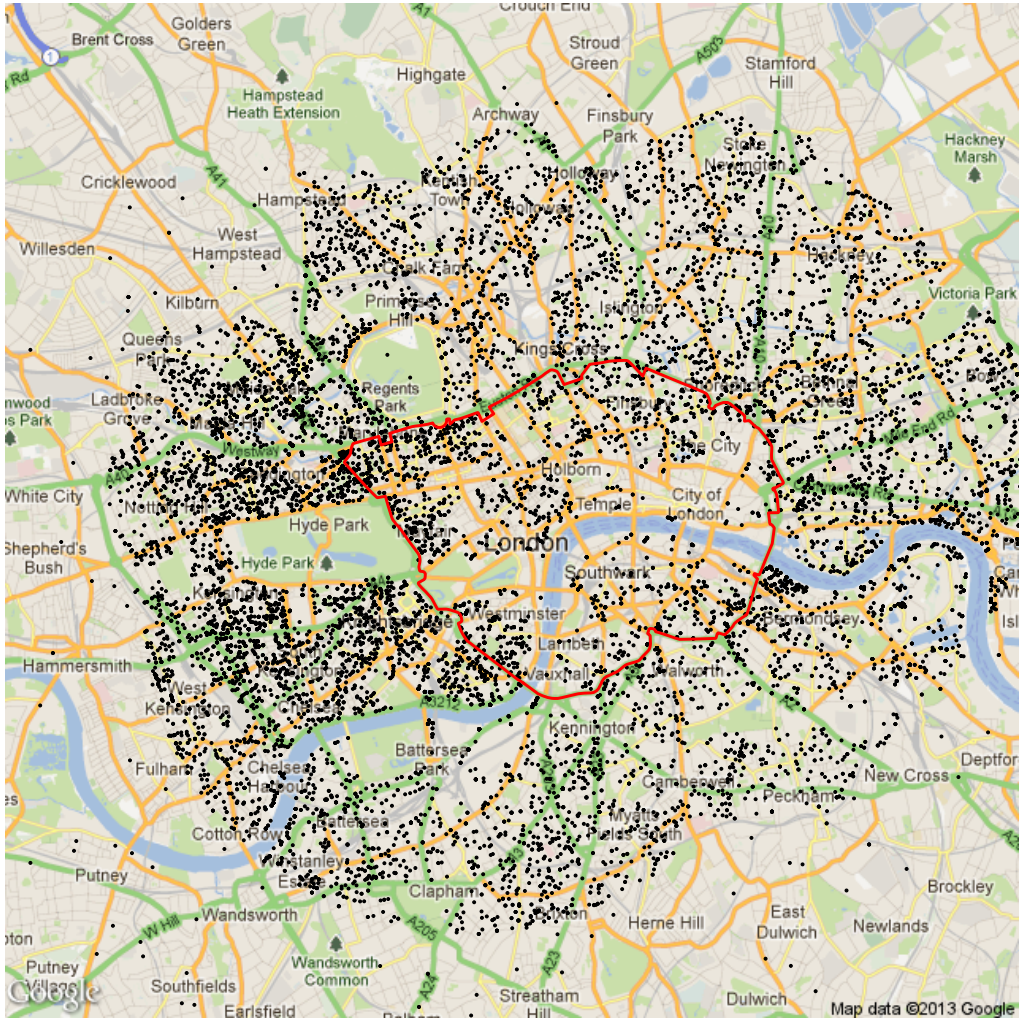
The World Wide Web is full of various kinds of new data, e.g.:

- open government data
- search engine data
- services that track social behavior

Practical arguments

- financial resources are sparse
- . . . and so is our time
- reproducibility

Real estate prices, London congestion charge



Data retrieved from http://www.zoopla.co.uk

Measuring issue salience using Wikipedia page view data

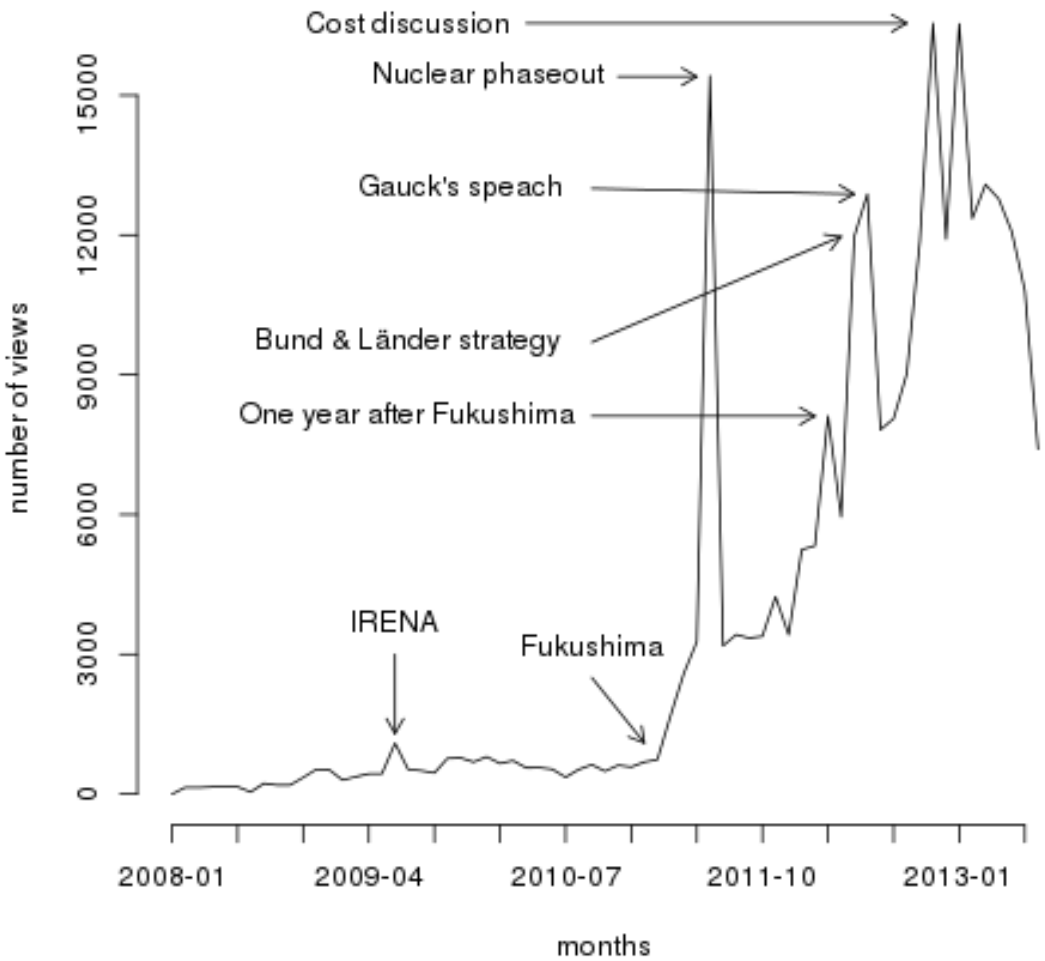


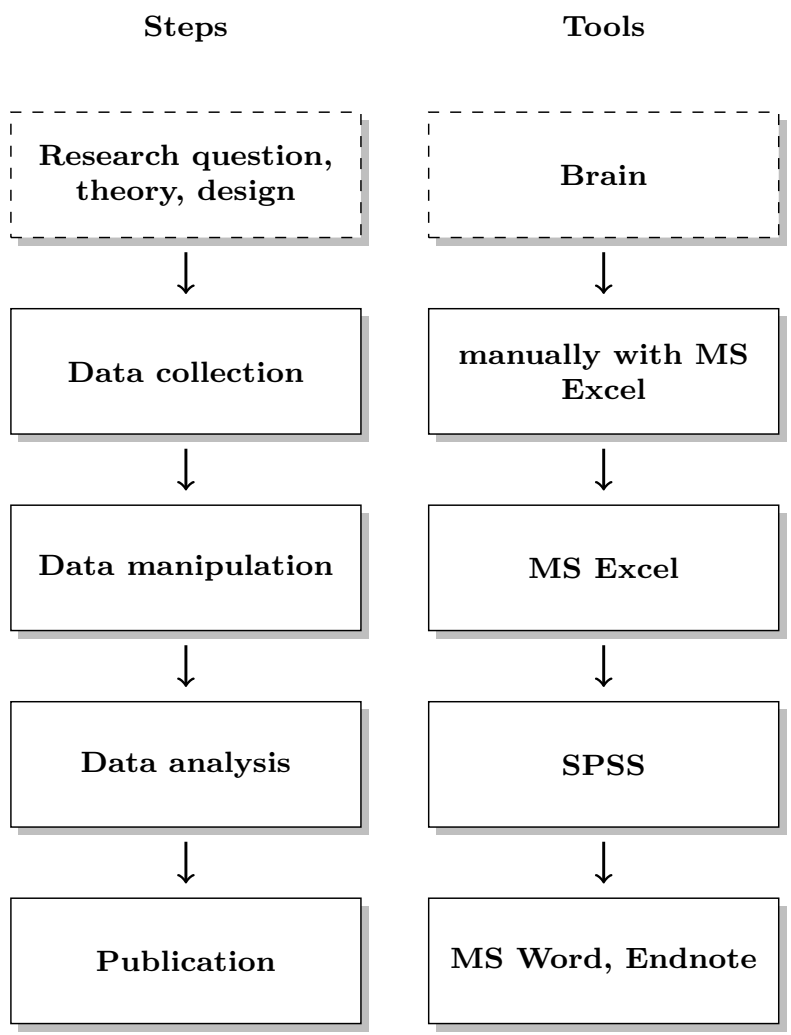
Figure 1: Wikipedia article views for "Energiewende" from January 2008 - July 2013

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

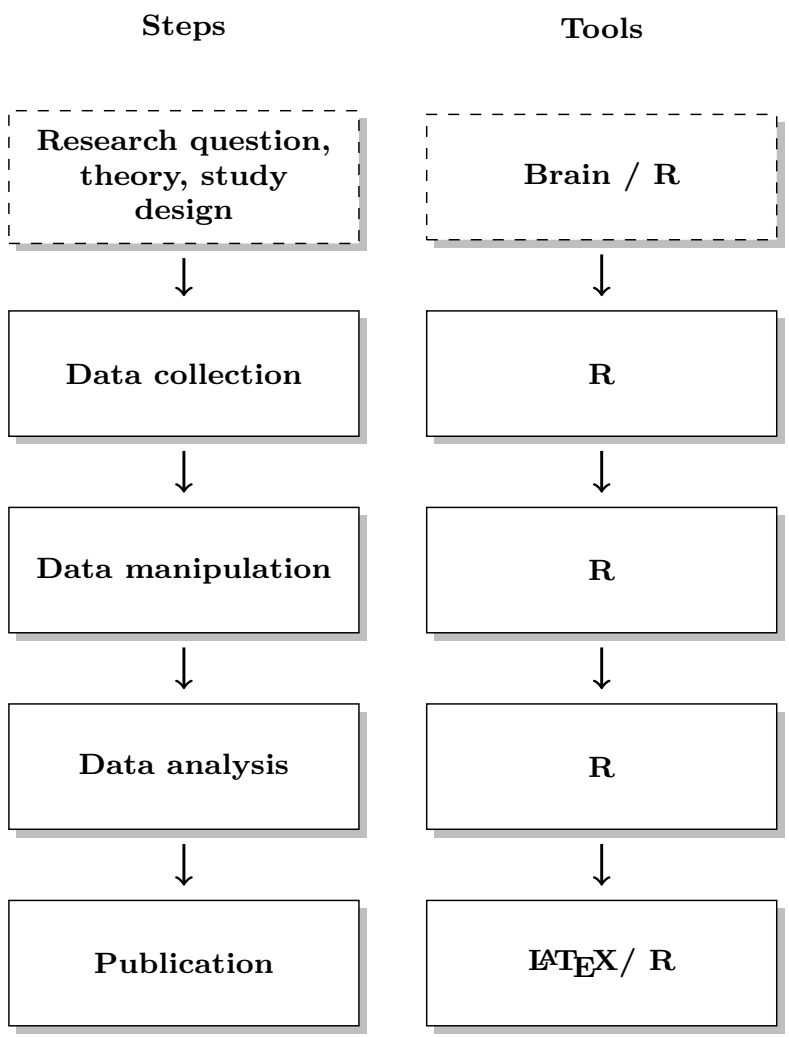
Why R?

- 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



Why R?

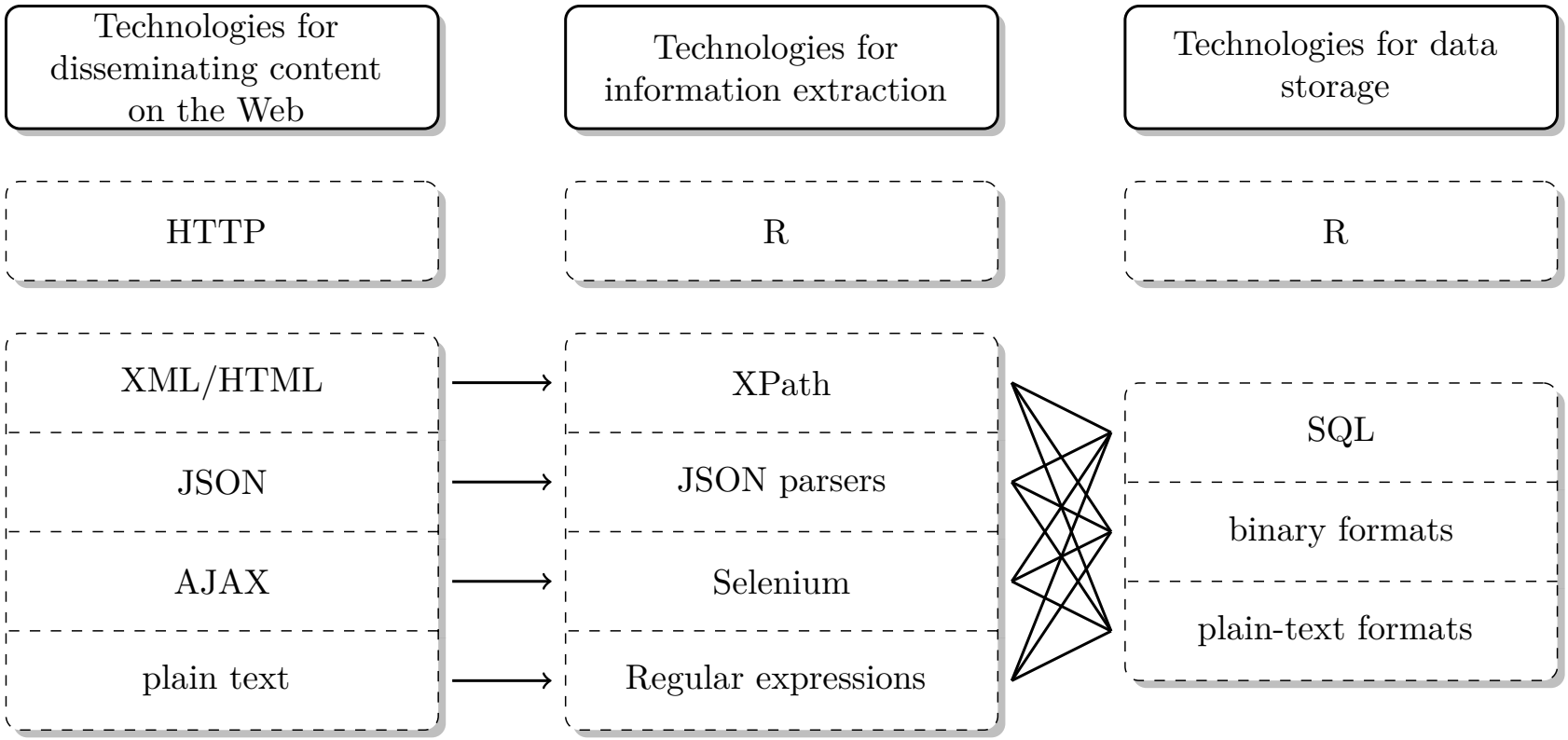
- free
- open source
- large community
- powerful tools for statistical analysis
- powerful tools for visualization
- flexible in processing all kinds of data/languages
- useful in every step of the workflow



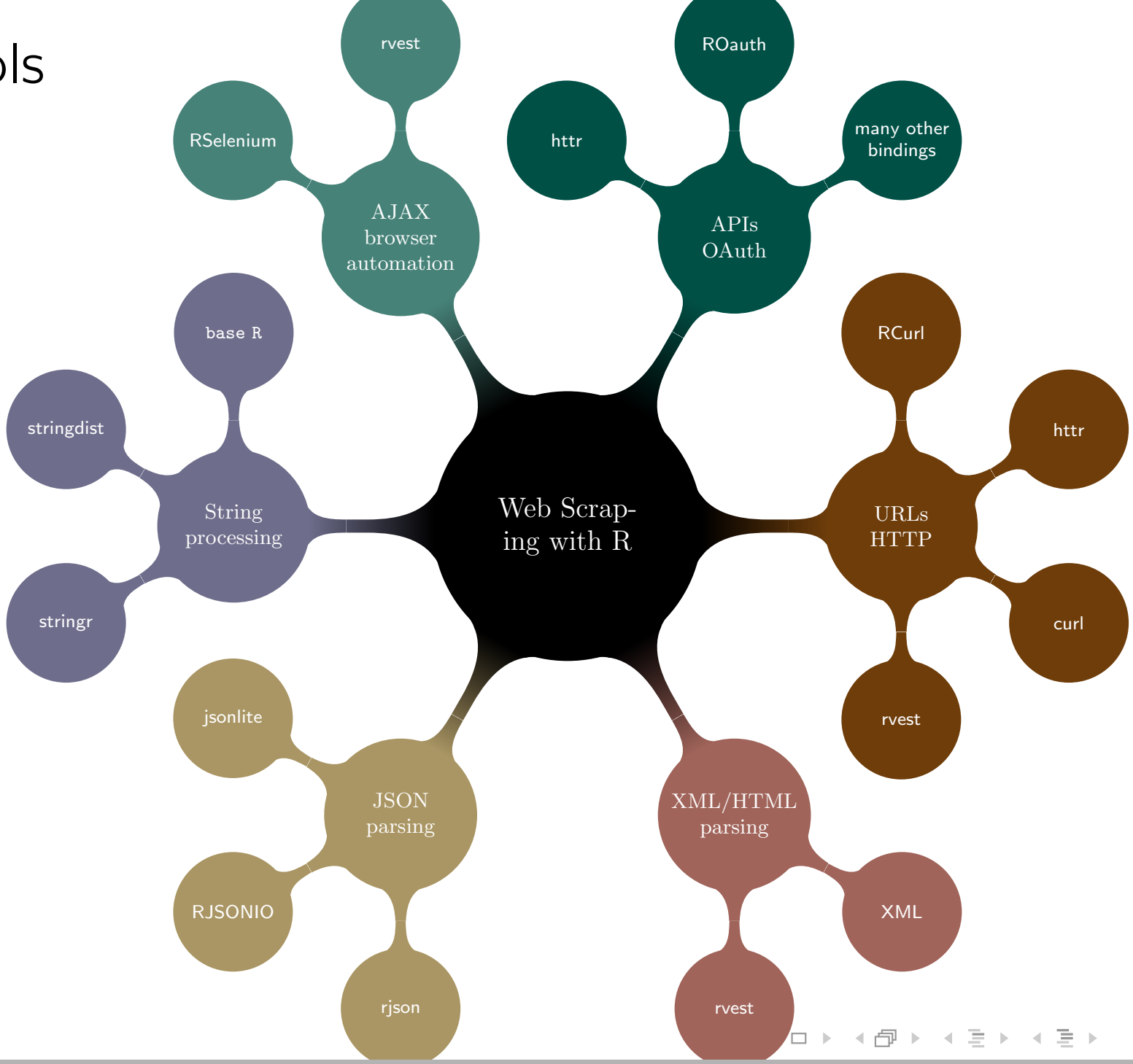
The philosophy behind web data collection with R

- no point-and-click procedure
- automation of download, parsing, and data extraction procedures
- classical screen scraping
- tapping of web services and APIs
- post-processing of text data
- reproducibility

Technologies of the World Wide Web



R tools


Web Scrap-
ing with R
AJAX
browser
automation
rvest
RSelenium
APIs
OAuth
ROauth
httr
many other
bindings
String
processing
base R
stringdist
stringr
URLs
HTTP
RCurl
httr
curl
rvest
JSON
parsing
jsonlite
RJSONIO
rjson
XML/HTML
parsing
XML
rvest

Technical Setup

1. make sure that the newest version of R (currently 3.1.2; available here) is installed on your computer
2. install the newest stable version of *RStudio* (available here)
3. install the following packages:
   ```
   pkgs <- c('RCurl', 'XML', 'stringr', 'jsonlite',
   'httr', 'rvest', 'devtools', 'RSelenium', 'plyr',
   'dpylr', 'wikipediatrend', 'twitteR', 'streamR')
   ```
4. install the *Chrome* (from here) and *Firefox* (from here) browsers
5. install *Java* (from here)

A first encounter with the Web using R

- Wickham's rvest package
- case study 1: mapping breweries in Germany
- case study 2: crafting name maps from phonebook data
- case study 3: building a network of political scientists

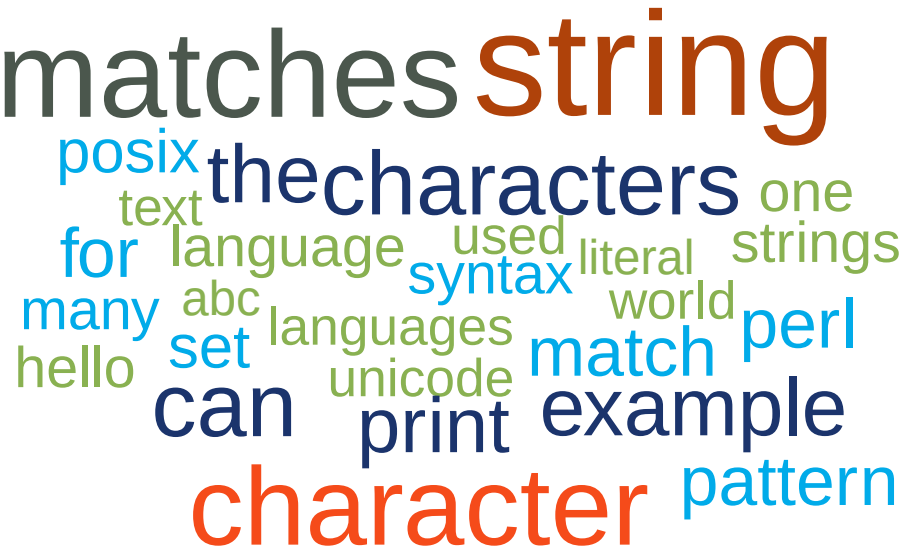
Exercises

1. Go to http://www.spiegel.de/schlagzeilen/ and inspect the page!
   1.1 Use SelectorGadget to find an XPath expression that covers all headlines!
   1.2 Write a little program in R that imports those headlines!
2. At http://en.wikipedia.org/wiki/List_of_European_Cup_and_UEFA_Champions_League_finals, you find some statistics on all Champions League finals.
   2.1 Find a function in the rvest package that is useful for scraping data from tables!
   2.2 Apply this function to retrieve the table 'List of European Cup and UEFA Champions League finals'!
   2.3 Based on the scraped data, identify the three nations with the most wins!

Part II

Regular Expressions

Overview



Regular Expressions
- What are regular expressions?
- Regular expressions in R
- Regex exercise
- String manipulation in R
- Character encodings
- String manipulation exercise
- Case study: mapping locations of AJPS reviewers

What are regular expressions?

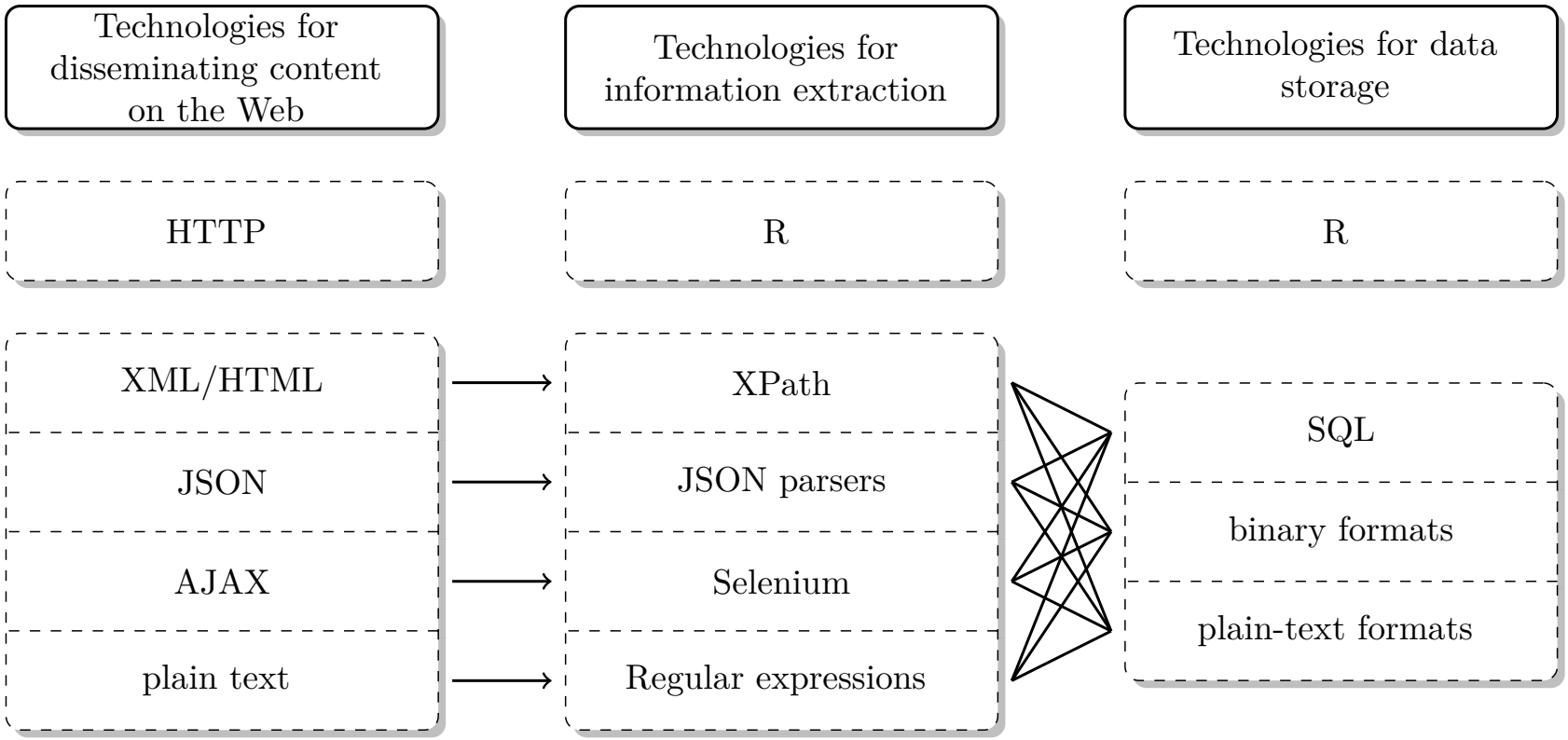
Definition

- 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 (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 usful for extraction and tidying purposes

Technologies of the World Wide Web



Introductory example

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

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

Introductory example

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

Procedure:

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

```
library(stringr)
name <- unlist(str_extract_all(raw.data, "[[:alpha:]., ]{2,}"))
name
```

```
## [1] "Moe Szyslak"
## [2] "Burns, C. Montgomery"
## [3] "Rev. Timothy Lovejoy"
## [4] "Ned Flanders"
## [5] "Simpson,Homer"
## [6] "Dr. Julius Hibbert"
```

Introductory example

Procedure, *continued*:

- construct regex for phone numbers
- apply regex on raw vector
- combine both vectors

```
phone <- unlist(str_extract_all(raw.data,
                "\\(?(\\d{3})?\\)?(-| )?\\d{3}(-| )?\\d{4}"))
phone
## [1] "555-1239"       "(636) 555-0113"
## [3] "555-6542"       "555 8904"
## [5] "636-555-3226"   "5553642"
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
```

Conclusion so far

- 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 useful under certain circumstances

Regular expressions in R

An example string:

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

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.

Regular expressions in R

Strings match themselves

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

Multiple matches are returned as a list

```
(out <- str_extract_all(c("text", "manipulation", "basics"), "a"))
## [[1]]
## character(0)
##
## [[2]]
## [1] "a" "a"
##
## [[3]]
## [1] "a"
```

Regular expressions in R

Character matching is case sensitive

```
str_extract(example.obj, "small")
## [1] "small"
str_extract(example.obj, "SMALL")
## [1] NA
str_extract(example.obj, ignore.case("SMALL"))
## [1] "small"
```

We can match arbitrary combinations of characters

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

Regular expressions in R

Matching the beginning of a string

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

Matching the ending of a string

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

Regular expressions in R

Pipe operator

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

The dot: the ultimate wildcard

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

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.

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

Regular expressions in R

The hyphen defines a range of characters

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

We can add additional characters to a character class

```
unlist(str_extract_all(example.obj, "[uvw. ]"))
##  [1] "." " " " " " " " "." " " " " " "." " " "
## [10] " " " " " "."
```

Regular expressions in R

Predefined character classes

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

```
unlist(str_extract_all(example.obj, "[[:punct:]]"))
## [1] "." "." "-" "." "."
unlist(str_extract_all(example.obj, "[:punct:]"))
##  [1] "n" "t" "n" "c" "n" "t" "t" "n" "n"
## [10] "t" "n" "c"
```

Regular expressions in R

Predefines 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

```
unlist(str_extract_all(example.obj, "[[:punct:]ABC]"))
## [1] "." "A" "." "-" "." "A" "."
unlist(str_extract_all(example.obj, "[^[:alnum:]]"))
##  [1] "." " " " " " " " " "." " " "-" " " " " "."
## [10] " " " " " " " " "."
```

Regular expressions in R

Alternative character classes

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

```
unlist(str_extract_all(example.obj, "\\w+"))
```

```
## [1] "1"        "A"        "small"
## [4] "sentence" "2"        "Another"
## [7] "tiny"     "sentence"
```

```
unlist(str_extract_all(example.obj, "e\\>"))
```

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

```
unlist(str_extract_all(example.obj, "e\\b"))
```

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

Regular expressions in R

Use of quantifiers

?	The preceding item is optional and will be matched at most once
*	The preceding item will be matched zero or more time
+	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

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

Regular expressions in R

Greedy quantification

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

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

Regular expressions in R

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?!)
- alternativly, use `fixed("symbols")` to let the parser interpret a chain of symbols literally

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

Regular expressions in R

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

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

Regular expressions in R

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 `(?<!...)`
- needs `perl` specification

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

Regular expressions in R

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)

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

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

Regular expressions in R

Backreferencing: a bit more complicated

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

Solution:

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

Mechanic:

1. match all letters without 'a': `[b-z]+`
2. the sequence should be a complete word with a beginning and an end: `\\<` and `\\>`
3. refer to the word: `()`
4. match anything in between: `.+?\\1`

Regex exercise

1. Describe the types of strings that conform to the following regular expressions and construct an example that is matched by the regular expression.:
   - 1.1 `[0-9]+\\$`
   - 1.2 `\\b[a-z]{1,4}\\b`
   - 1.3 `.*?\\.txt$`
   - 1.4 `\\d{2}/\\d{2}/\\d{4}`
   - 1.5 `<(.+?)>.+?</\\1>`
2. Consider the mail address chunkylover53[at]aol[dot]com.
   - 2.1 Transform the string to a standard mail format using regular expressions.
   - 2.2 Imagine we are trying to extract the digits in the mail address using `[:digit:]`. Explain why this fails and correct the expression.

String manipulation in R

Why string manipulation

- text processing
- important operations: extraction of text patterns, data tidying, preparation of text corpora for statistical text processing and, of course, screen scraping

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

Functions of the stringr package

Function	Description	Output
Functions using regular expressions		
`str_extract()`	Extracts first string that matches pattern	Character vector
`str_extract_all()`	Extracts all strings that match pattern	List of character vectors
`str_locate()`	Return position of first pattern match	Matrix of start/end positions
`str_locate_all()`	Return positions of all pattern matches	List of matrices
`str_replace()`	Replaces first pattern match	Character vector
`str_replace_all()`	Replaces all pattern matches	Character vector
`str_split()`	Split string at pattern	List of character vectors
`str_split_fixed()`	Split string at pattern into fixed number of pieces	Matrix of character vectors
`str_detect()`	Detect pattern in string	Boolean vector
`str_count()`	Count number of pattern occurrences in string	Numeric vector
Further functions		
`str_sub()`	Extract strings by position	Character vector
`str_dup()`	Duplicate strings	Character vector
`str_length()`	Length of string	Numeric vector
`str_pad()`	Pad a string	Character vector
`str_trim()`	Discard string padding	Character vector
`str_c()`	Concatenate strings	Character vector

String manipulation in R

String localization

```
str_locate(example.obj, "tiny")
##      start end
## [1,]    35  38
```

Substring extraction

```
str_sub(example.obj, start = 35, end = 38)
## [1] "tiny"
```

String replacement

```
str_sub(example.obj, 35, 38) <- "huge"
str_replace(example.obj, pattern = "huge", replacement = "giant")
## [1] "1. A small sentence. - 2. Another giant sentence."
```

String manipulation in R

String splitting

```
unlist(str_split(example.obj, "-"))
```

```
## [1] "1. A small sentence. "
## [2] " 2. Another huge sentence."
```

String splitting with fixed number of elements

```
as.character(str_split_fixed(example.obj, "[[:blank:]]", 5))
```

```
## [1] "1."
## [2] "A"
## [3] "small"
## [4] "sentence."
## [5] "- 2. Another huge sentence."
```

String manipulation in R

Manipulation of several elements

- until this point we applied functions to vectors of length one
- however, it is common to apply functions to multi-element vectors

Example object with several elements:

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

String detection

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

String manipulation in R

String counting

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

String duplication

```
str_dup(char.vec, 3)
## [1] "thisthisthis"
## [2] "and thisand thisand this"
## [3] "and thatand thatand that"
```

String manipulation in R

String joining

```
str_c("text", "manipulation", sep = " ")
## [1] "text manipulation"
```

```
cat(str_c(char.vec, collapse = "\n"))
## this
## and this
## and that
```

```
str_c("text", c("manipulation", "basics"), sep = " ")
## [1] "text manipulation"
## [2] "text basics"
```

String manipulation in R

Approximate matching

- Matching of approximately equal strings, (e.g., "Österreich" and "Osterreich")
- in principle, we could program naive matching algorithms using regex
- better: use of more powerful algorithms, e.g. Levenshtein distance
- `agrep()` function in base R
- more extended functionality provided by the stringdist package

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

Beware: character encodings

- character encoding schemes define how certain characters are represented in binary information
- text data come in various encodings, e.g., ASCII or unicode (UTF-8 and others)
- web-based text data usually come in UTF-8
- when working on a Windows machine, the system encoding scheme is, by default, Windows-1525
- if a program (e.g., R) falsely assumes a certain encoding scheme, the result is often a mess

```
small.frogs <- "Små grodorna, små grodorna är lustiga att se."
Encoding(small.frogs) <- "windows-1252"
small.frogs.utf8
```

```
SmA  grodorna, smA grodorna Ar lustiga att se.
```

Beware: character encodings

How to deal with character encodings

- indicate encoding in advance
- use converter functions

```
htmlParse(url, encoding = "utf-8")
html(url, encoding = "utf8")
```

```
small.frogs <- "Små grodorna, små grodorna är lustiga att se."
small.frogs
## [1] "Små grodorna, små grodorna är lustiga att se."
small.frogs.utf8 <- iconv(small.frogs, from = "windows-1252", to = "UTF-8")
Encoding(small.frogs.utf8)
## [1] "UTF-8"
small.frogs.utf8
## [1] "Små grodorna, små grodorna är lustiga att se."
```

String manipulation exercise

1. Consider the following (adapted) poem by Robert Gernhardt!
   - 1.1 Restore the original poem by deleting all large and small 'L's from the text!
   - 1.2 Extract all capital letters and collapse them into one word!
   - 1.3 Replace every word that contains three or less characters with "bla"!

```
poem <- c(
  "Am Tag, an dem das L verschwand,",
  "da war die Luft voll Klagen.",
  "Den Dichtern, ach, verschlug es glatt",
  "ihr Singen und ihr Sagen.",
  "Nun gut. Sie haben sich gefasst.",
  "Man sieht sie wieder schreiben.",
  "Jedoch:",
  "Solang das L nicht wiederkehrt,",
  "muß alles Flickwerk beiben.")
```

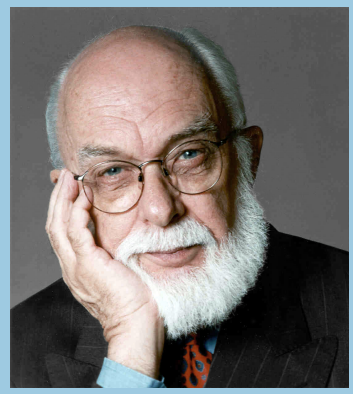
Case study: mapping locations of AJPS reviewers

Goal: map locations of AJPS reviewers

- fetch list of AJPS reviewers from PDFs
- locate them on a map

Tasks:

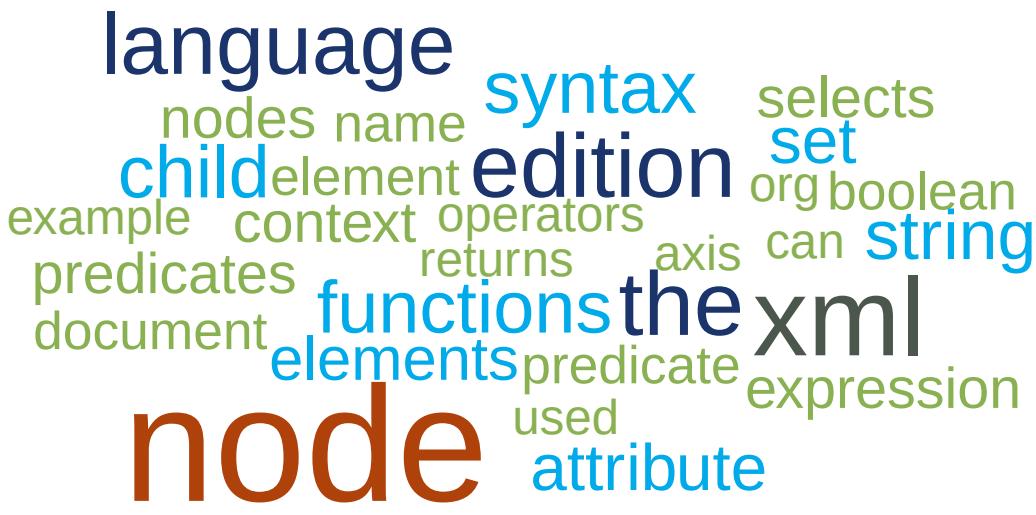
- download PDF files from http://ajps.org/list-of-reviewers/
- import them into R (as plain text)
- extract information via regular expressions
- geocoding



Part III

HTML and XPath

Overview



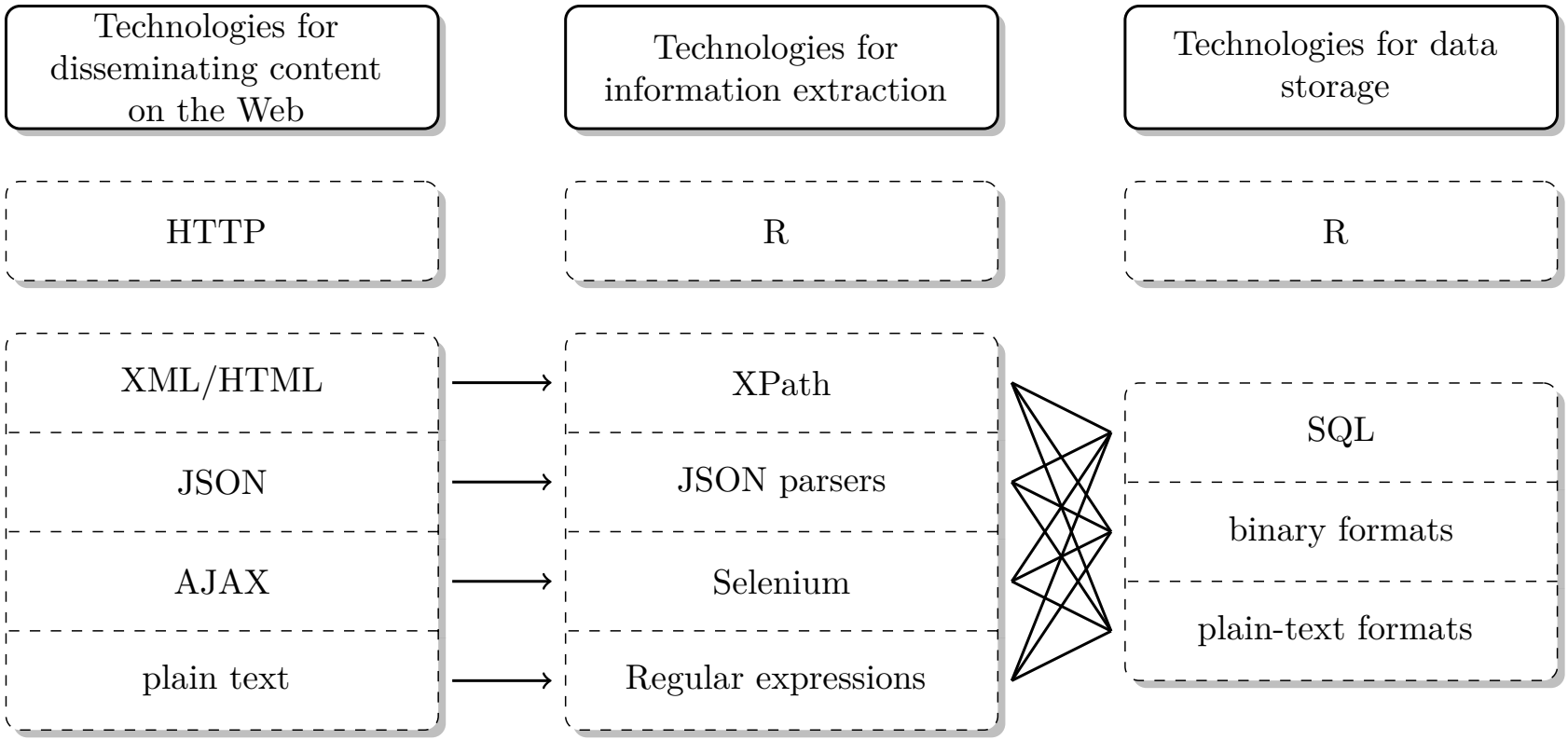
HTML and XPath
- HTML – a quick primer
  - HTML syntax
  - Important tags and attributes
- What's XPath?
- XPath in R
  - Grammar of XPath
  - Identification of node sets
  - Extraction of Node Elements
  - Making XPath expressions more flexible
- Case study: generating name maps from phonebook entries

HTML – a quick primer

What's HTML?

- **H**yper**T**ext **M**arkup **L**anguage
- markup language = plain text + markups
- standard for the construction of websites
- relevance for web scraping: web architecture is important because it determines where and how information is stored

Technologies of the World Wide Web



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:

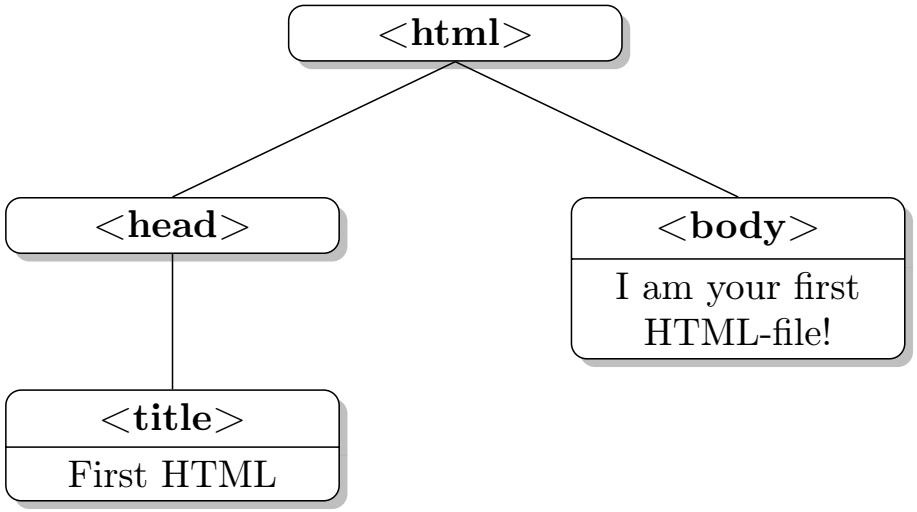
```
1 <a href="http://www.r-datacollection.com/">Link to Homepage</a>
```

Syntax

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

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>
```



Important tags and attributes

- 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:

```
<a href="en.wikipedia.org/wiki/List\_of\_lists\_of\_lists">Link with absolute path</a>
```

Reference in a document:

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

Link to a reference:

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

Important tags and attributes

Meta data tag <meta>

- empty tag in the `head` element
- specifies meta information like author and encoding

Definition of key words:

```
<meta name="keywords" content="Automation, Data, R">
```

Rules for bots:

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

Info on character encoding:

```
<meta charset="ISO-8859-1"/>
```

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:

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

```
<h1>heading of level 1 -- this will be BIG</h1>
...
<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 <li>Dogs</li>
3 <li>Cats</li>
4 <li>Fish</li>
5 </ul>
```

Important tags and attributes

Organizational tags and

- grouping of content over lines ( ) or within lines ()
- 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

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>

- construction of tables

Example:

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

What's XPath?

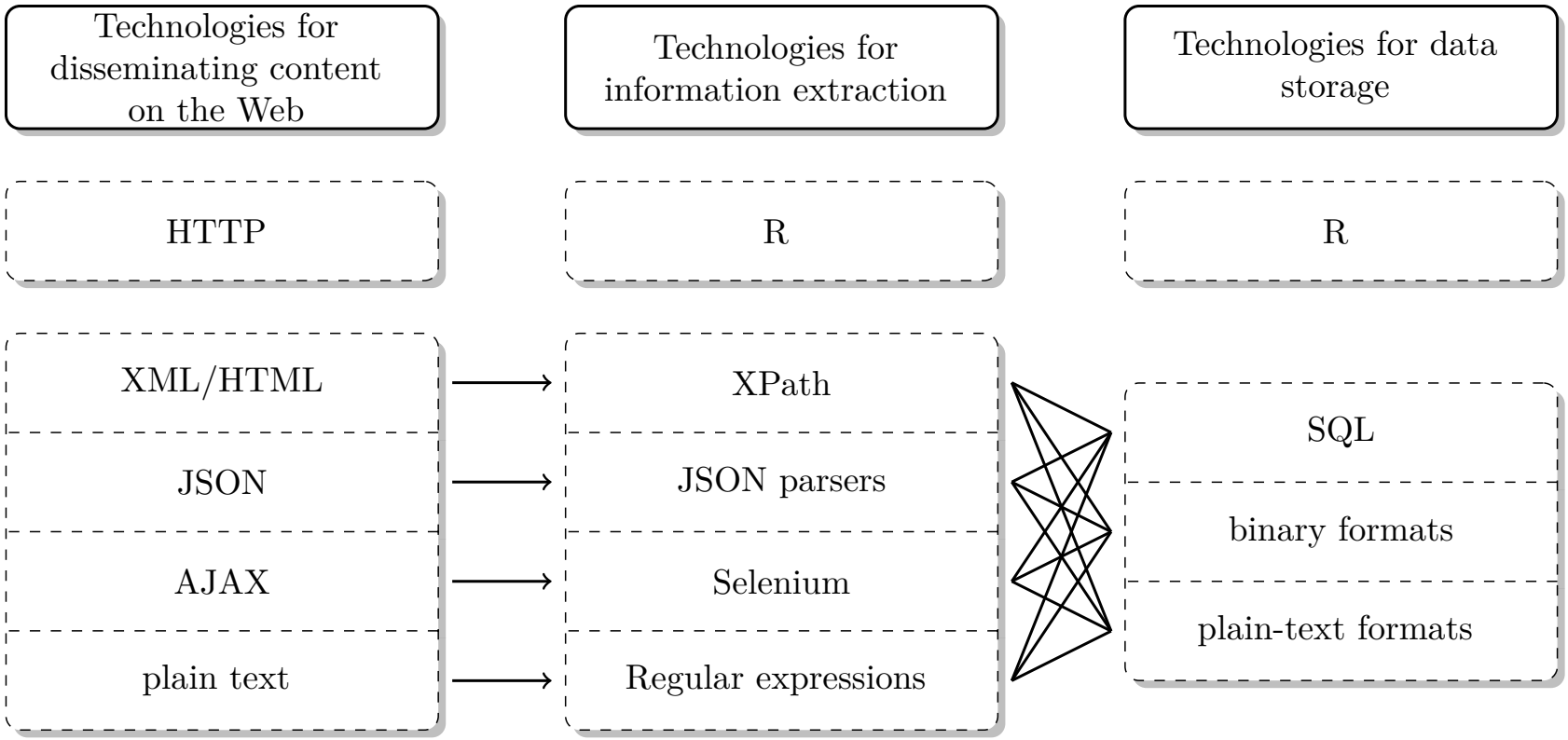
Definition

- XML Path language, a W3C standard
- Query language for XML-based documents (i.e., for HTML as well)
- access node sets and extract content

Why XPath for web scraping?

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

Technologies of the World Wide Web



Example

Procedure:

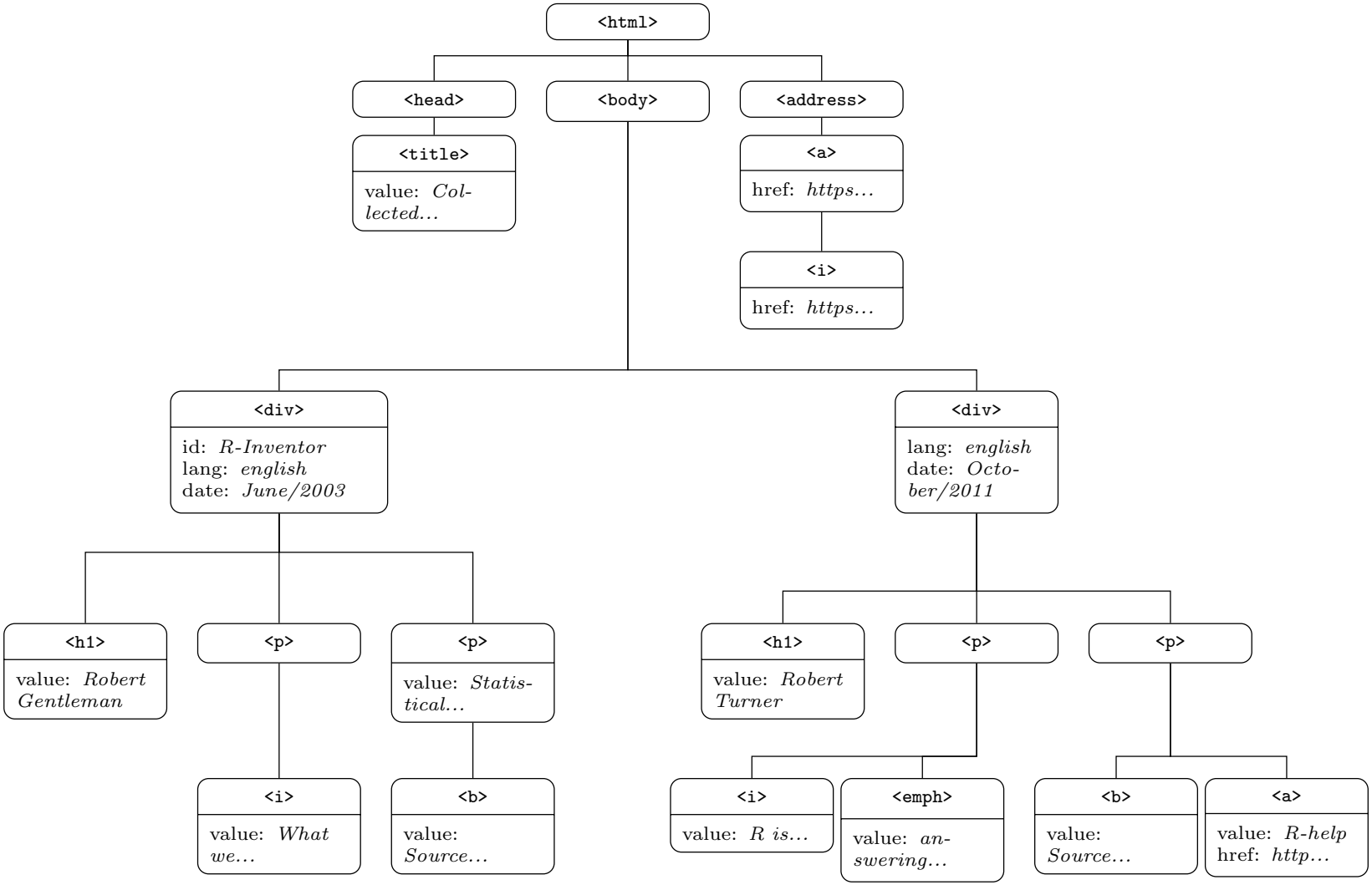
- load package XML
- parse document
- query document with XPath
- XML can process XPath queries

```
library(XML)
```

```
parsed_doc <- htmlParse(file = "../materials/fortunes.html")
```

```
xpathSApply(parsed_doc, "//div[last()]/p/i", xmlValue)
## [1] "'R is wonderful, but it cannot work magic'"
```

```
print(parsed_doc)
## <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
## <html>
## <head><title>Collected R wisdoms</title></head>
## <body>
## <div id="R Inventor" lang="english" date="June/2003">
##   <h1>Robert Gentleman</h1>
##   <p><i>'What we have is nice, but we need something very different'</i></p>
##   <p><b>Source: </b>Statistical Computing 2003, Reisensburg</p>
## </div>
## <div lang="english" date="October/2011">
##   <h1>Rolf Turner</h1>
##   <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering a request for
##   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p
## </div>
## <address>
## <a href="http://www.r-datacollection.com"><i>The book homepage</i></a><a></a>
## </address>
## </body>
## </html>
##
```


<html>
<head>
<body>
<address>
<title>
value: Col-
lected...
<a>
href: https...
<i>
href: https...

id: R-Inventor
lang: english
date: June/2003

lang: english
date: Octo-
ber/2011
<h1>
value: Robert
Gentleman
<p>
<p>
value: Statis-
tical...
<h1>
value: Robert
Turner
<p>
<p>
<i>
value: What
we...
<b>
value:
Source...
<i>
value: R is...
<emph>
value: an-
swering...
<b>
value:
Source...
<a>
value: R-help
href: http...

Grammar of XPath

Example:

```
xpathSApply(doc=parsed_doc, path="/html/body/div/p/i")
## [[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>
```

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 `xpathSApply()` function

Grammar of XPath

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, '//')

```
xpathSApply(doc=parsed_doc, path="/html/body/div/p/i")
## [[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>
```

```
xpathSApply(parsed_doc, "//body//p/i")
## [[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>
```

Grammar of XPath

```
xpathSApply(parsed_doc, "//i")
## [[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>
```

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

Grammar of XPath

Wildcard operator

- meta symbol $*$
- matches any node
- far less important than wildcards in regular expressions

```
xpathSApply(parsed_doc, "/html/body/div/*/i")
## [[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>
```

It does not work that way:

```
xpathSApply(parsed_doc, "/html/body/*/i")
## list()
## attr(,"class")
## [1] "XMLNodeSet"
```

Grammar of XPath

Navigational operators '.' and '..'

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

```
xpathSApply(parsed_doc,"//title/..")
## [[1]]
## <head>
##   <title>Collected R wisdoms</title>
## </head>
```

Grammar of XPath

Pipe operator

- combines several paths

```
xpathSApply(parsed_doc, "//address | //title")
## [[1]]
## <title>Collected R wisdoms</title>
##
## [[2]]
## <address>
##   <a href="http://www.r-datacollection.com">
##     <i>The book homepage</i>
##   </a>
##   <a/>
## </address>
```

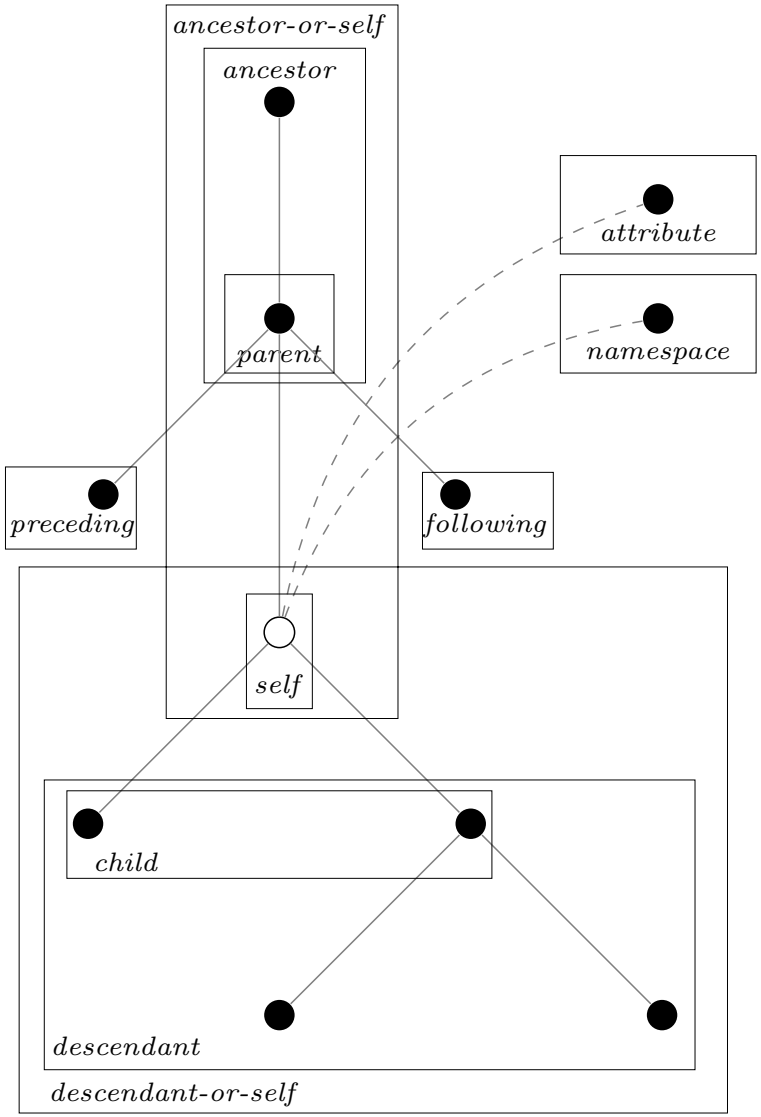
Grammar of XPath

Multiple queries at once

- compact way of pipe operator

```
twoQueries <- c(address="//address", title="//title")
xpathSApply(parsed_doc, twoQueries)
## [[1]]
## <title>Collected R wisdoms</title>
##
## [[2]]
## <address>
##   <a href="http://www.r-datacollection.com">
##     <i>The book homepage</i>
##   </a>
##   <a/>
## </address>
```

Node relations in XPath



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 ending

Node relations in XPath

'Family relations' between nodes

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

```
xpathSApply(parsed_doc, "//a/ancestor::div")
## [[1]]
## <div lang="english" date="October/2011">
##   <h1>Rolf Turner</h1>
##   <p><i>'R is wonderful, but it cannot work magic'</i> <br/><emph>answering a request fo
##   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p>
## </div>
```

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

‘Family relations’ between nodes

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

s

```
xpathSApply(parsed_doc, "//p/preceding-sibling::h1")
## [[1]]
## <h1>Robert Gentleman</h1>
##
## [[2]]
## <h1>Rolf Turner</h1>
```

Predicates

Predicates

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

```
node[predicate]
```

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

Predicates

Numeric predicates

- Exploitation of positions, counts, etc.

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

```
xpathSApply(parsed_doc, "//div/p[position()=1]")
## [[1]]
## <p>
##   <i>'What we have is nice, but we need something very different'</i>
## </p>
##
## [[2]]
## <p><i>'R is wonderful, but it cannot work magic'</i> <br/><emph>answering a request for a
```

Predicates

Numeric predicates

Example:Select all `<div>` nodes which have at least one `descendant` node of type `<a>`.

```
xpathSApply(parsed_doc, "//div[count(.//a)>0]")
## [[1]]
## <div lang="english" date="October/2011">
##   <h1>Rolf Turner</h1>
##   <p><i>'R is wonderful, but it cannot work magic'</i> <br/><emph>answering a request fo
##   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p
## </div>
```

Predicates

Numeric predicates

Further examples:

```
xpathSApply(parsed_doc, "//div/p[last()-1]")
xpathSApply(parsed_doc, "//div[count(./@*)>2]")
xpathSApply(parsed_doc, "//*[string-length(text())>50]")
```

Boolean function `not()`

```
xpathSApply(parsed_doc, "//div[not(count(./@*)>2)]")
```

Predicates

Textual predicates

- exploitation of text features
- applicable on: node name, content, , attributes, attribute values
- XPath 2.0 supports (simplified) regex, however, R (the XML package) does not support XPath 2.0

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

```
xpathSApply(parsed_doc, "//div[@date='October/2011']")
## [[1]]
## <div lang="english" date="October/2011">
##   <h1>Rolf Turner</h1>
##   <p><i>'R is wonderful, but it cannot work magic'</i> <br/><emph>answering a request fo
##   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p>
## </div>
```

Predicates

Partial matching

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

```
xpathSApply(parsed_doc, "//*[contains(text(), 'magic')]")
## [[1]]
## <i>'R is wonderful, but it cannot work magic'</i>
```

Further examples:

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

Extraction of node elements

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
`xmlName`		node name
`xmlValue`		node value
`xmlGetAttr`	`name`	node attribute
`xmlAttrs`		(all) node attributes
`xmlChildren`		node children
`xmlSize`		node size

Extraction of node elements

Values

```
xpathSApply(parsed_doc, "//title", fun = xmlValue)
## [1] "Collected R wisdoms"
```

Attributes

```
xpathSApply(parsed_doc, "//div", xmlAttrs)
## [[1]]
##           id         lang         date
## "R Inventor"    "english"  "June/2003"
##
## [[2]]
##           lang           date
##      "english" "October/2011"
```

Attribute values

```
xpathSApply(parsed_doc, "//div", xmlGetAttr, "lang")
## [1] "english" "english"
```

Extraction of node elements

Expanding the `fun` argument

- flexible processing of node sets
- adaptation of the `fun` argument of the `xpathSApply()` function
- useful scenarios: data tidying, exception handling, error handling

Example: Make nodes' content lower case.

```
lowerCaseFun <- function(x){
  x <- tolower(xmlValue(x))
  x
  }
```

Application:

```
xpathSApply(parsed_doc, "//div//i", fun = lowerCaseFun)
## [1] "'what we have is nice, but we need something very different'"
## [2] "'r is wonderful, but it cannot work magic'"
```

Extraction of node elements

Expanding the `fun` argument

Example: *'Extract years from attributes'*

```
dateFun <- function(x){
  require(stringr)
  date <- xmlGetAttr(node = x, name = "date")
  year <- str_extract(date, "[0-9]{4}")
  year
}
```

Application:

```
xpathSApply(parsed_doc, "//div", dateFun)
## [1] "2003" "2011"
```

Extraction of node elements

Expanding the `fun` argument

Example: *'Exception handling for `NULL` objects'*

```
idFun <- function(x){
  id <- xmlGetAttr(x, "id")
  id <- ifelse(is.null(id), "not specified" , id)
  return(id)
}
```

Application:

```
xpathSApply(parsed_doc, "//div", idFun)
## [1] "R Inventor"    "not specified"
```

Making XPath expressions more flexible

Variables in XPath expressions

- sometimes a single XPath expression is not enough to access all desired nodes
- solution 1: manually generate several XPath expressions and proceed sequentially
- solution 2: use XPath with variables. Workhorse: `sprintf()` function

`sprintf()` allows variables within strings:

```
sprintf("Kevin-%s Schmidt", c("Prince", "Hayden", "Justin"))
## [1] "Kevin-Prince Schmidt"
## [2] "Kevin-Hayden Schmidt"
## [3] "Kevin-Justin Schmidt"
sprintf("%s-%s Schmidt", c("Heinz", "Gustav"), c("Prince", "Hayden"))
## [1] "Heinz-Prince Schmidt"
## [2] "Gustav-Hayden Schmidt"
sprintf("%s %d", "Klasse", 1:3)
## [1] "Klasse 1" "Klasse 2" "Klasse 3"
```

More info on variable types: `?sprintf`

Making XPath expressions more flexible

Example:

```
parsed_stocks <- xmlParse(file = "../materials/technology.xml")
companies <- c("Apple", "IBM", "Google")
```

Use `sprintf()`:

```
(expQuery <- sprintf("//%s/close", companies))
## [1] "//Apple/close"  "//IBM/close"
## [3] "//Google/close"
```

Construct extractor function:

```
getClose <- function(node){
  value <- xmlValue(node)
  company <- xmlName(xmlParent(node))
  mat <- c(company = company, value = value)
}
```

Making XPath expressions more flexible

Example, *continued*:

Apply XPath query:

```
stocks_extracted <- xpathSApply(parsed_stocks, expQuery, getClose)
stocks_extracted[,1:3]
```

```
##         [,1]      [,2]     [,3]
## company "Apple"   "Apple"  "Apple"
## value   "520.634" "520.01" "519.048"
```

Convert into `data.frame`:

```
stocks <-as.data.frame(t(stocks_extracted))
stocks$value <- as.numeric(as.character(stocks$value))
head(stocks, 3)
```

```
##   company   value
## 1   Apple 520.634
## 2   Apple 520.010
## 3   Apple 519.048
```

Namespaces in XPath expressions

Namespaces

- used as prefix for text to cope with different XML flavors or sources within a document
- XPath operates on the standard namespace

```
parsed_xml <- xmlParse("../materials/titles.xml")
parsed_xml
## <?xml version="1.0" encoding="UTF-8"?>
## <!DOCTYPE presidents SYSTEM "presidents.dtd">
## <root xmlns:h="http://www.w3.org/1999/xhtml" xmlns:t="http://funnybooknames.com/crockfor
##   <h:head>
##     <h:title>Basic HTML Sample Page</h:title>
##   </h:head>
##   <t:book id="1">
##     <t:author>Douglas Crockford</t:author>
##     <t:title>JavaScript: The Good Parts</t:title>
##   </t:book>
## </root>
##
```

Namespaces in XPath expressions

XPath fails if we do not take namespaces into account:

```
xpathSApply(parsed_xml,  "//title", fun=xmlValue)
## list()
```

Bypassing namespaces:

```
xpathSApply(parsed_xml, "//*[local-name()='title']", xmlValue)
## [1] "Basic HTML Sample Page"
## [2] "JavaScript: The Good Parts"
```

Explicit use of namespaces:

```
xpathSApply(parsed_xml, "//x:title", namespaces = c("x" = "http://funnybooknames.com/crockf
## [1] "JavaScript: The Good Parts"
```

Accessing namespace definitions:

```
nsDefs <- xmlNamespaceDefinitions(parsed_xml)[[2]]
ns  <- nsDefs$uri
parsed_xml <- xpathSApply(parsed_xml, "//x:title", namespaces = c("x" = ns), xmlValue)
```

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

No.

XPath 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

Case study: generating name maps from phonebook entries

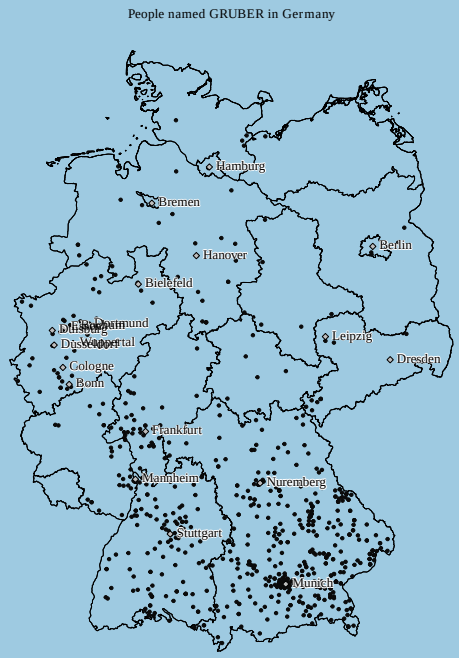
Goal: build name maps from phonebook entries

- fetch entries from an online phonebook
- extract addresses
- locate entries on a map

Tasks:

- inspect http://www.dastelefonbuch.de/
- develop scraping strategy
- apply strategy: retrieve data, extract information, cleanse data
- visualize / analyze data
- generalize scraping task

More details in the book, Chapter 15



Exercises

1. http://www.transparency.org/ is the webpage of Transparency International, an anti-corruption NGO.
   - 1.1 Store the page in a local folder!
   - 1.2 Parse the page from the local file!
   - 1.3 Construct an XPath expression that extracts all hyperlinks on the homepage and apply it on the parsed page!
   - 1.4 Construct an XPath expression that extracts all the headlines in the "highlights" section!
   - 1.5 Collect the category names that are listed under the "Media" field at the bottom of the page!
2. Go to http://en.wikipedia.org/wiki/List_of_MPs_elected_in_the_United_Ki and extract the table containing the elected MPs int the United Kingdom general election of 1992. Which party has most Sirs?

Part IV

JSON

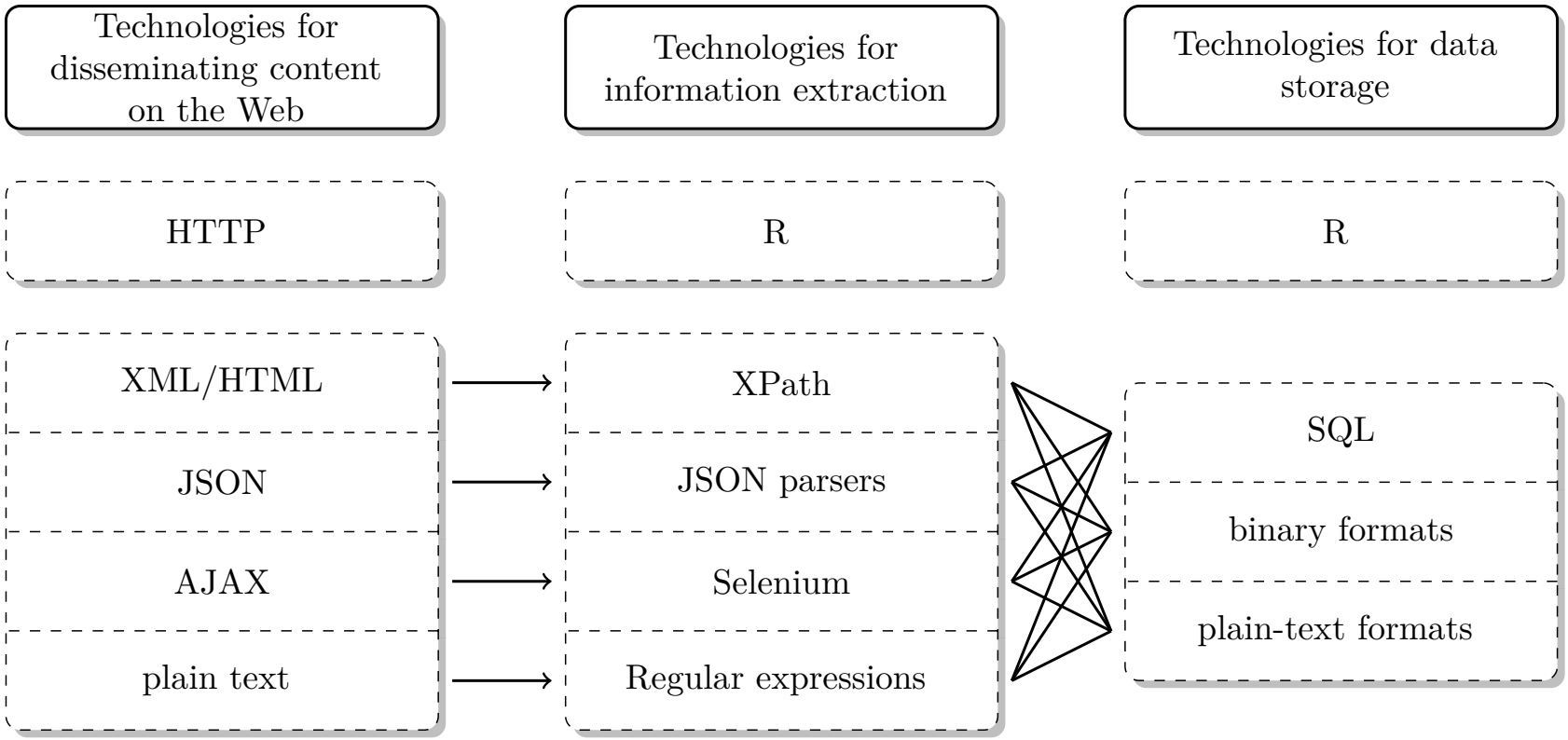
Overview

JSON
- What's JSON?
- JSON syntax
- JSON and R
- Case study: Wikipedia page view statistics
- Exercises

What's JSON?

- **J**ava**S**cript **O**bject **N**otation
- 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

Technologies of the World Wide Web



Example

```
{"indy movies" :[
  {"name" : "Raiders of the Lost Ark",
  "year" : 1981,
"actors" : {
"Indiana Jones": "Harrison Ford",
"Dr. Rene Belloq": "Paul Freeman"
},
"producers": ["Frank Marshall", "George Lucas", "Howard Kazanjian"],
"budget" : 18000000,
"academy_award_ve": true},
{"name" : "Indiana Jones and the Temple of Doom",
"year" : 1984,
"actors" : {
"Indiana Jones": "Harrison Ford",
"Mola Ram": "Amish Puri"
},
"producers": ["Robert Watts"],
"budget" : 28170000,
"academy_award_ve": true}
  ]
}
```

Brackets

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.

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

2. Keys and values are separated by a colon.

```
"year" : 1981
```

3. Key-value pairs are separated by commas.

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

4. Values within arrays are separated by commas.

```
["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 isomophic 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	`true` or `false`
null	`null`, an unknown value
Object	content in curly brackets
Array	ordered content in square brackets

JSON vs. XML

Is JSON the better XML?

- JSON – "the fat-free alternative to XML" (json.org)
- JSON does **not** provide natively: comments feature, namespaces, internal validation syntax, labeling of objects, document markup
- JSON offers: tight syntax, immediate mapping of data structure in many programming languages

JSON and R

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

Mapping rules of jsonlite

```
library(jsonlite)
x <- '[1, 2, true, false]'
fromJSON(x)
## [1] 1 2 1 0
x <- '["foo", true, false]'
fromJSON(x)
## [1] "foo"   "TRUE"  "FALSE"
x <- '[1, "foo", null, false]'
fromJSON(x)
## [1] "1"     "foo"   NA      "FALSE"
```

JSON and R

Parsing with jsonlite

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

JSON and R

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

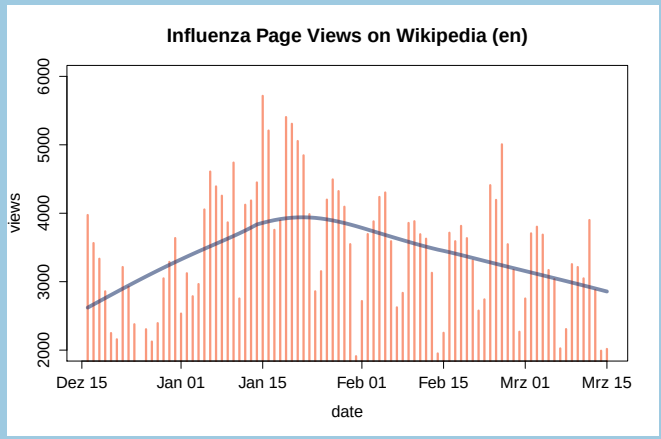
Case study: Wikipedia page view statistics

Goal: fetch Wikipedia page view statistics

- download JSON data
- explore trends

Tasks:

- access statistics (JSON format) from http://stats.grok.se
- import data into R
- tidy data
- explore trends



Exercises

1. The file 'oscartweets.json' provides some data on tweets that were posted during the 2014 Academy Awards ceremony.
   - 1.1 Import the data into R and try to figure out what information the dataset contains! Which R object classes do the elements of the parsed object have?
   - 1.2 Extract the tweets from the dataset and store them into a character vector!
   - 1.3 Construct a dataset which contains some user information, and append the tweet itself and its date of creation!

Part V

APIs

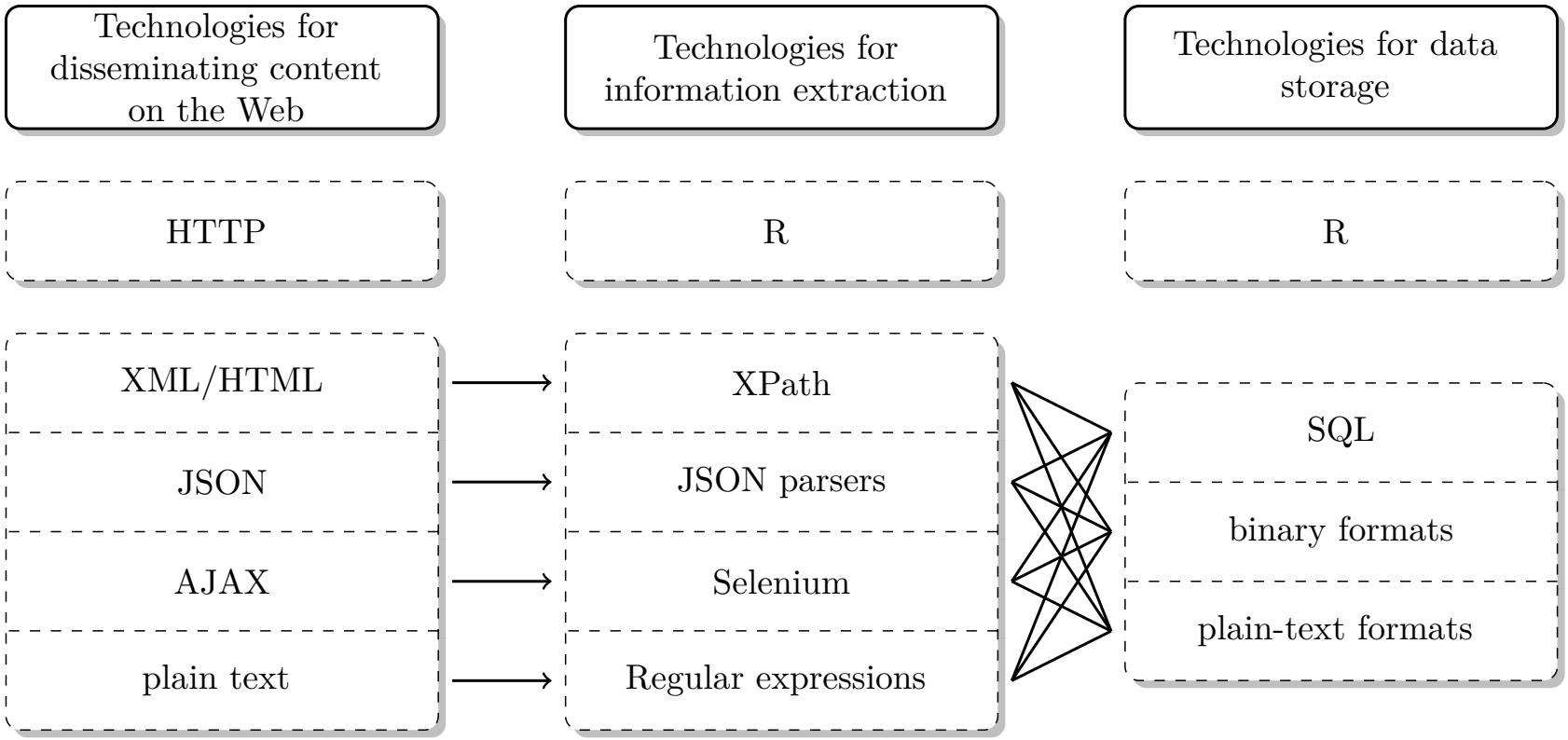
Overview

APIs
- What are APIs?
- Accessing APIs with R
- Case study: Wikipedia page views, reconsidered
- Exercises
- Social media mining with R
- Case study: exploring Twitter's services
- Exercises

What are APIs?

- **A**pplication **P**rogramming **I**nterface
- many web services provide APIs to access their data and services (Twitter, Google, Facebook, Wikipedia, . . .)
- instant access to clean data
- frees us from building manual scrapers
- forces us to understand the API architecture
- common data formats: XML, JSON

Technologies of the World Wide Web



Data gathering with APIs

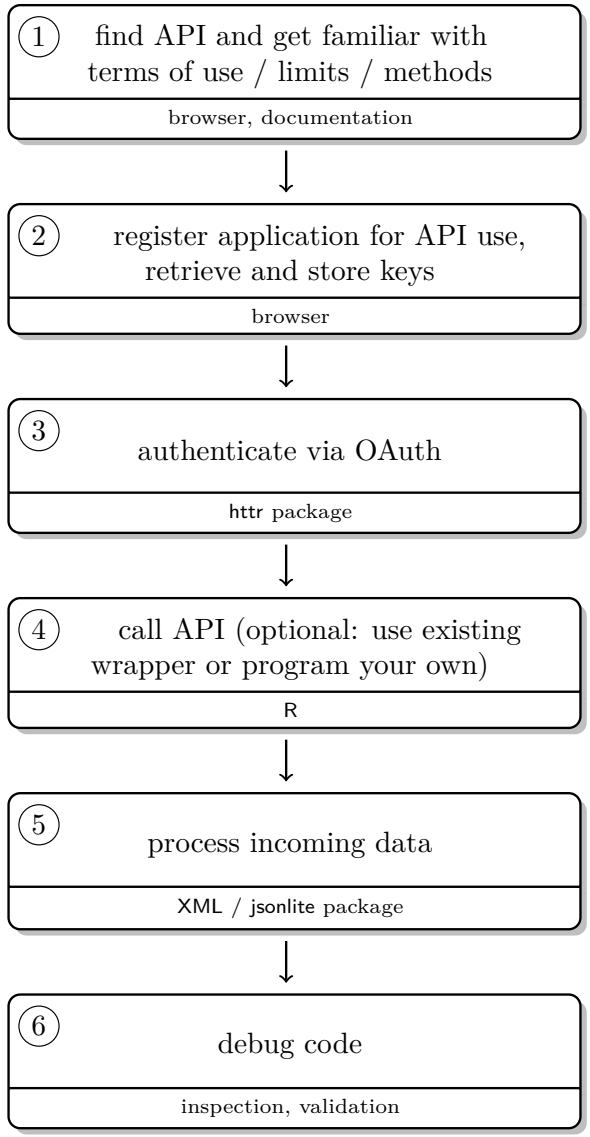
Advantages

- pure data collection without 'layout waste'
- standardized data access
- de facto automatic agreement of data owner
- robustness of calls

Disadvantages

- requires knowledge of API architecture
- dependent upon API suppliers
- not always for free

Data gathering with APIs



Finding APIs on the Web

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

rOpenSci: Collection of R-API interfaces:
http://ropensci.org/

CRAN Task View of Web Technologies:
http://cran.r-project.org/web/views/WebTechnologies.html

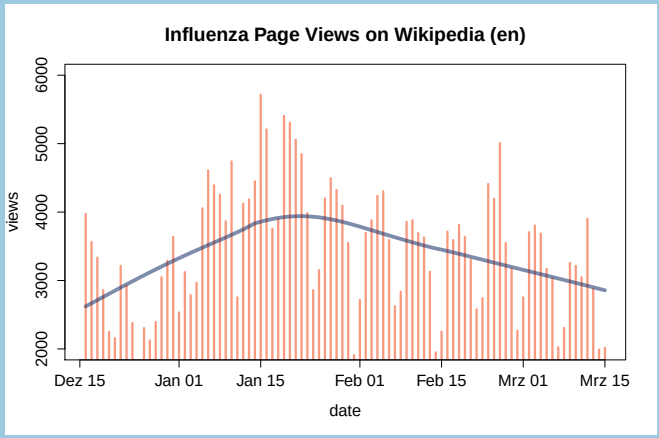
Some API resources

- Google Maps
  - http://maps.googleapis.com/maps/api/directions/json?origin=Konstanz,Germany1&destination=Bamberg
  - https://developers.google.com/maps/documentation/directions/
- GitHub
  - https://api.github.com/users/simonmunzert
  - https://developer.github.com/v3/
- Twitter
  - https://api.twitter.com/1.1/statuses/user_timeline.json
  - https://dev.twitter.com/overview/documentation

Case study: Wikipedia page views, reconsidered

Goal: fetch Wikipedia page view statistics

- automate the process
- construct high-level functions that facilitate multiple requests



Exercises

1. Yahoo! provides a free weather RSS feed that enables you to get up-to-date weather information for any location.
   - 1.1 Explore the API at https://developer.yahoo.com/weather/documentation.html!
   - 1.2 The API requires a WOEID to identify the location. Try to find a source on the Web that returns a WOEID for a written location name (e.g., 'Berlin').
   - 1.3 Access the API with R and transform the data into a useful format, e.g., a data.frame object!

Social media mining with R

Why social media mining?

- network data
- communication data
- preference data

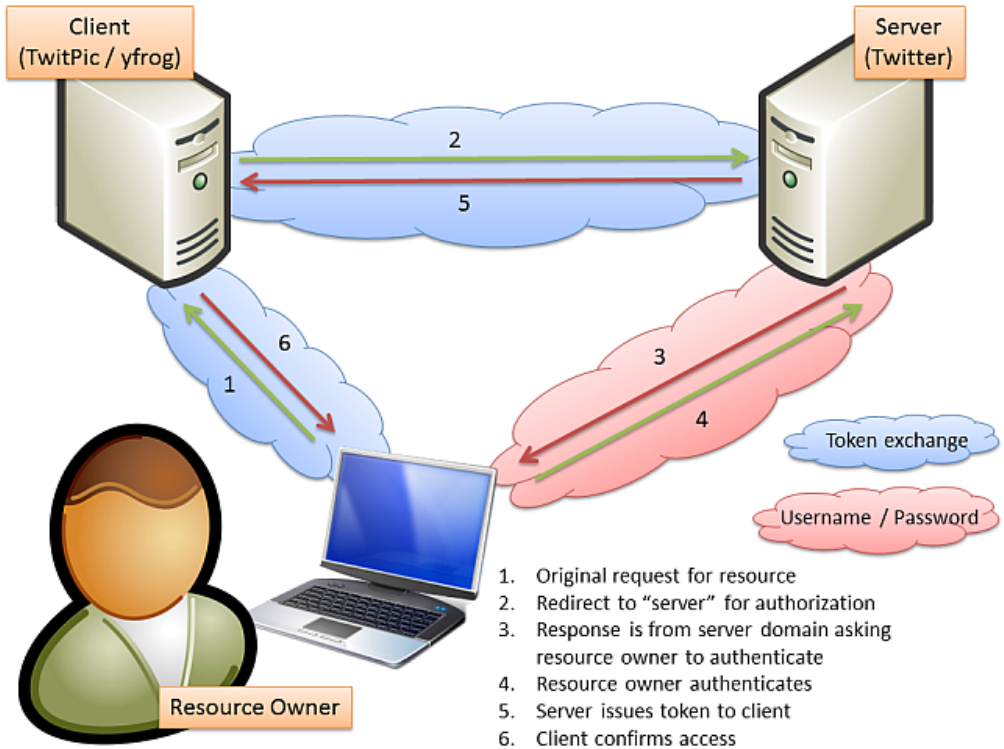
Existing R bindings

- twitteR
- streamR
- Rfacebook
- Rlinkedin
- SocialMediaMineR
- tumblR
- ...

Accessing APIs with OAuth authorization

What's OAuth?

- authorization standard
- used to provide client applications access to owner's resources



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

Accessing APIs with OAuth authorization

The OAuth workflow with httr

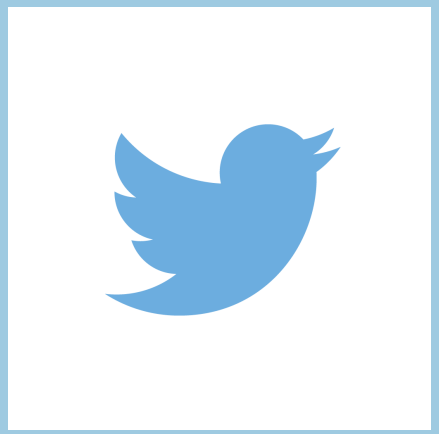
- `oauth_endpoint()`: define Oauth endpoints for the request and access token
- `oauth_app()`: bundle consumer key and secret to request access credentials
- `oauth1.0_token()` and `oauth2.0_token()`: exchange consumer key and secret for access key and secret
- `sign_oauth1.0()` and `sign_oauth2.0()`: create signature from received access token

... but sometimes the R API binding simplifies matters (e.g., the newest version of the twitteR package)

Case study: exploring Twitter's services

Goal: tap Twitter's REST and Streaming APIs
Tasks:

- register app
- manage authorization process
- get to know the twitteR and the streamR packages



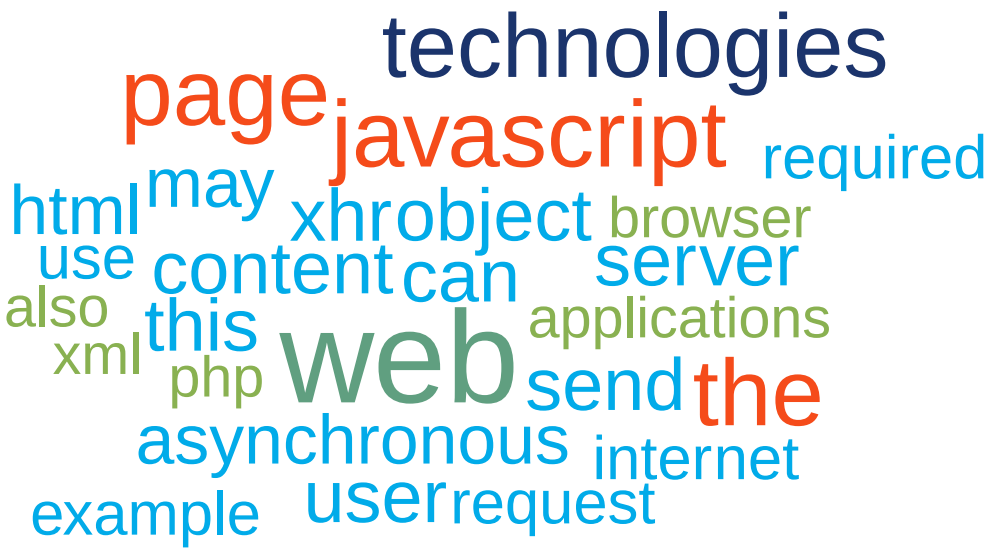
Exercises

1. At https://twitter.com/phdwhipbot you find a Twitter bot that is entirely run by R.
   1.1 Familiarize yourself with the mechanics of the bot at http://www.r-datacollection.com/blog/Programming-a-Twitter-bot/!
   1.2 Try to come up with a bot of your own and get it up and running!

Part VI

AJAX and Selenium

Overview



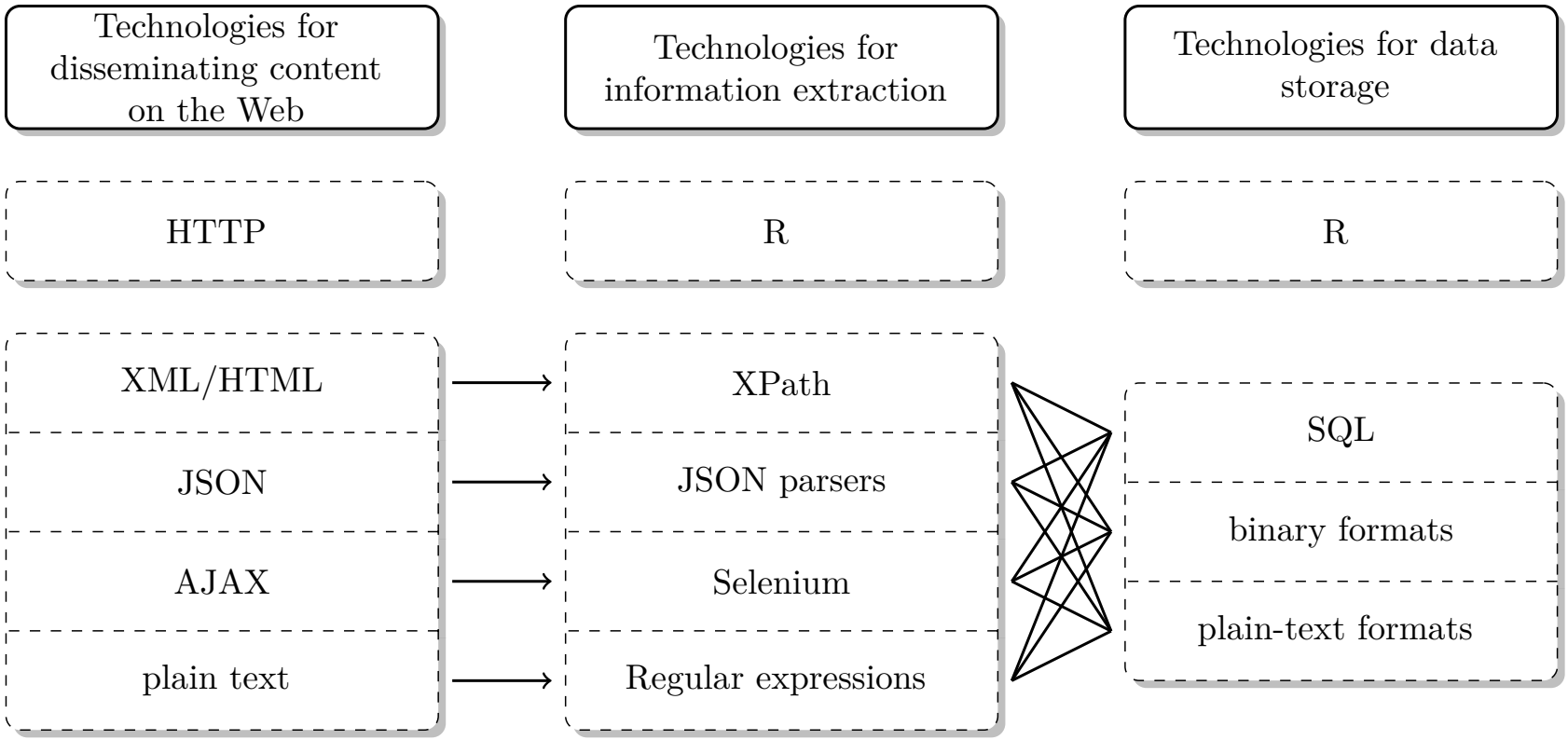
AJAX and Selenium
- What's AJAX?
- JavaScript
- XHR
- Selenium

What's AJAX?

- HTML/HTTP are used for static display of content
- 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
- **A**synchronous **J**avaScript **a**nd **X**ML' is a set of technologies that serve these purposes
- massively used in modern webpage design and architecture
- makes classical screen scraping more difficult

Example: https://twitter.com/regsprecher

Technologies of the World Wide Web



JavaScript

What's JavaScript?

- Programming language that connects well to web technologies
- W3C web standard
- native browser support
- extensible by many libraries
- *jQuery* library for DOM manipulation

JavaScript on the Web

How's JavaScript code embedded in HTML?

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

JavaScript on the Web

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

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

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

<head>
<title>Collected R wisdoms</title>
</head>

<body>
<div id="R Inventor" lang="english" date="June/2003">
  <h1>Robert Gentleman</h1>
  <p><i>'What we have is nice, but we need something very different'</i></p>
  <p><b>Source: </b>Statistical Computing 2003, Reisensburg</p>
</div>

<div lang="english" date="October/2011">
  <h1>Rolf Turner</h1>
  <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering a request for automatic generation of '
        data from a known mean and 95% CI'</emph></p>
  <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p>
</div>

<address><a href="http://www.r-datacollection.com"><i>The book homepage</i></a></address>
</body>
</html>
```

JavaScript on the Web

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 execuption 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

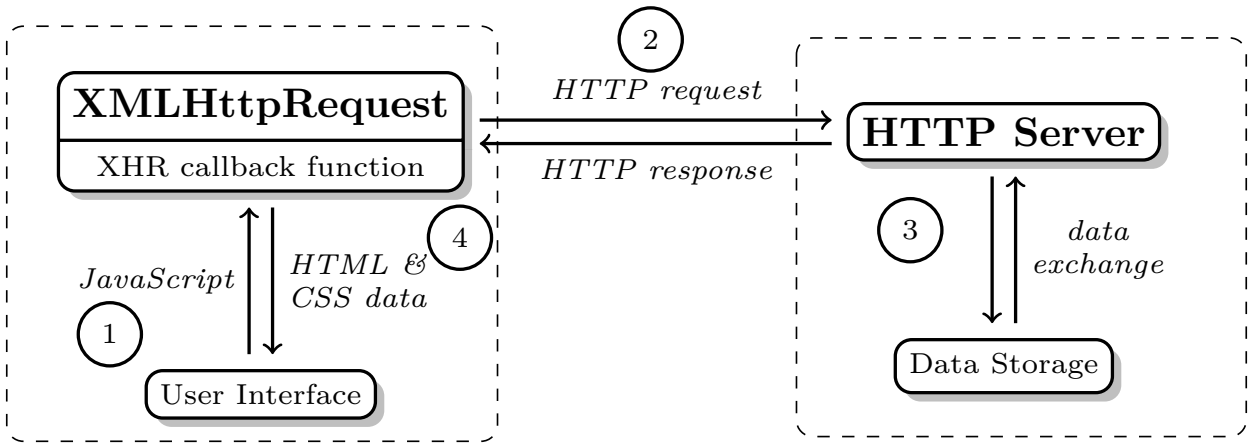
JavaScript on the Web

```
library(XML)
(fortunes1 <- htmlParse("../materials/fortunes1.html"))
## <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
## <html>
## <head>
## <script type="text/javascript" src="jquery-1.8.0.min.js"></script><script type="text/java
## </head>
## <body>
## <div id="R Inventor" lang="english" date="June/2003">
##   <h1>Robert Gentleman</h1>
##   <p><i>'What we have is nice, but we need something very different'</i></p>
##   <p><b>Source: </b>Statistical Computing 2003, Reisensburg</p>
## </div>
##
## <div lang="english" date="October/2011">
##   <h1>Rolf Turner</h1>
##   <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering a request for
##   <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p>
## </div>
##
## <address><a href="http://www.r-datacollection.com"><i>The book homepage</i></a></address>
## </body>
## </html>
##
```

XHR

What's XHR?

- **X**ML**H**ttp**R**equest
- interface for dynamic HTTP client-server communication
- classic HTTP: synchronous, XHR: asynchronous



XHR

Example: dynamic import of HTML

```
library(XML)
(fortunes2 <- htmlParse("../materials/fortunes2.html"))
## <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
## <html>
## <head>
## <script type="text/javascript" src="jquery-1.8.0.min.js"></script><script type="text/java
## </head>
## <body>
## <address><a href="http://www.r-datacollection.com"><i>The book homepage</i></a></address>
## </body>
## </html>
##
```

http://www.r-datacollection.com/materials/ajax/

XHR

Example: processing of JSON data

```
1  $("#quoteButton").click(function(){
2      $.getJSON("quotes/all_quotes.json", function(data){
3          $.each(data, function(key, value){
4              $("body").prepend("<div date='"+value.date+"'><h1>"+value.author+"</h1><
                   p><i>'"+value.quote+"'</i></p><p><b>Source: </b>'"+value.source+"
                   '</p></div>");
5          });
6      });
7  });
```

http://www.r-datacollection.com/materials/ajax/

Selenium

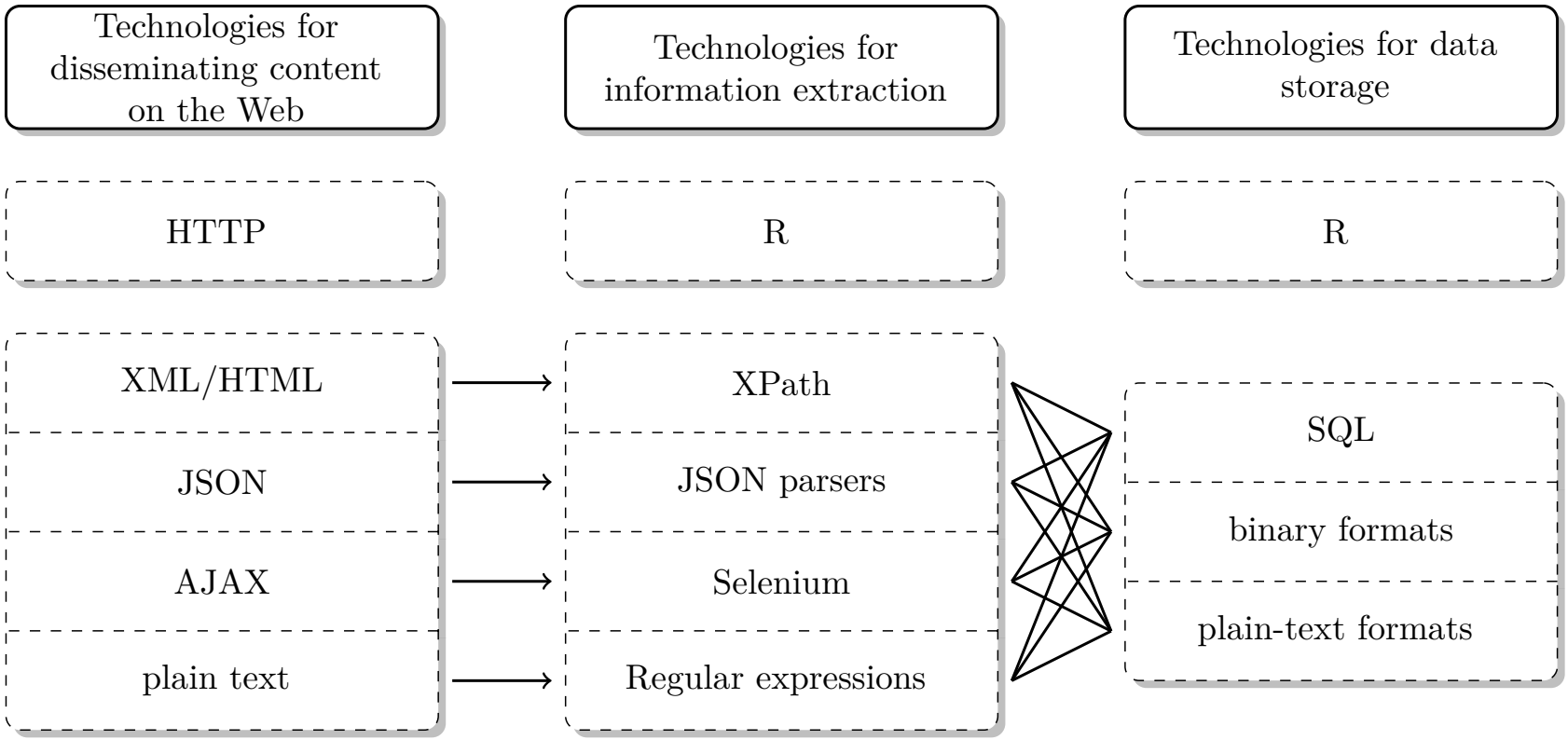
The problem reconsidered

- dynamic data requests are not stored in the static HTML page
- therefore, we cannot access them with classical methods and packages (httr, XML, `download.file()`, etc.)

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

Technologies of the World Wide Web



Selenium

What's Selenium?

- http://www.seleniumhq.org
- free software environment for automated web application testing
- several modules for different tasks; most important for our purposes: Selenium WebDriver
- Selenium WebDiver starts a server instance (as proxy) and passes commands (posed in R in our case) to the browser
- automated browsing via scripts

Selenium and R

Software requirements

- Java, https://www.java.com/de/download/
- Selenium server, http://selenium-release.storage.googleapis.com/2.45/selenium-server-standalone-2.45.0.jar or via RSelenium and `checkForServer()`
- Firefox browser, https://www.mozilla.org/en-US/firefox/new/
- RSelenium package

Case study: tapping the IEA Global Renewable Energy database

Goal: fetch policy data from IEA database

- pose request, retrieve output
- store the data in an R data.frame

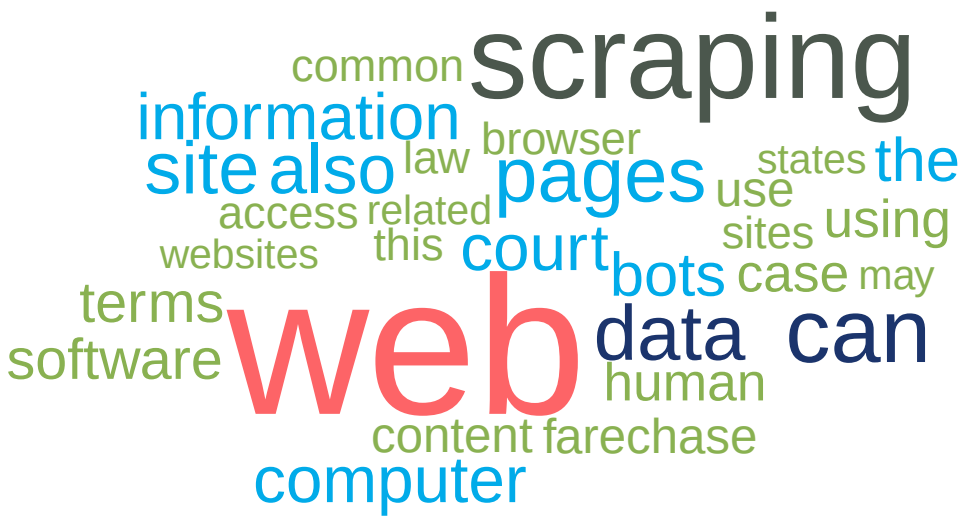
Tasks:

- get Selenium running
- inspect HTML form on http://www.iea.org/policiesandmeasures/renewableenergy/
- access page with RSelenium
- download data output
- import data into R
- data tidying

Part VII

Advanced Scraping Applications

Overview



Advanced Scraping Applications
- How URLs work
- Scraping data with URL manipulation
- Case study: scraping JStatSoft download statistics
- How HTML forms work
- Case study: Checking texts on readability
- Exercises

How URLs work

URLs

- **U**niform **R**esource **L**ocators
- main way of accessing resources on servers
- important component of client-server communication

Generic URL scheme

scheme://hostname:port/path?querystring

Example:

http://en.wikipedia.org:80/w/index.php?title=Special%3ASearch&profile=default&search=bruce&fulltext=Search

URL syntax

URLs dismantled

- **scheme**: which protocol to use, e.g., `http`, `ftp` or `file`
- **hostname**: which server to contact
- **port**: which port to contact
- **path**: which resource to request
- **query string**: which additional parameters to use; `name=value`, separated by &

URL syntax

URL encoding

- URLs are encoded in ASCII
- all special characters and characters not covered in ASCII have to be escaped
- → URL encoding or 'percent' encoding

```
t <- "I'm Eddie! How are you & you? 1 + 1 = 2"
(url <- URLencode(t, reserve = TRUE))
## [1] "I%27m%20Eddie%21%20How%20are%20you%20%26%20you%3f%201%20%2b%201%20%3d%202"
```

```
URLdecode(url)
## [1] "I'm Eddie! How are you & you? 1 + 1 = 2"
```

Scraping data with URL manipulation

Why URL-based web scraping?

- often, data are scattered across several pages
- directories of a website follow specific systematics
- we can 'manually' build locators of the resources by constructing a bunch of URLs

Procedure:

1. identify the running mechanism in the URL syntax
2. retrieve links to the running pages
3. download running pages
4. optional: retrieve additional information from running pages; continue with download etc.

Case study: scraping JStatSoft download statistics

Goal: examine download statistics of book review articles of the Journal of Statistical Software

- download HTML pages
- extract bibliometrical information
- (download PDFs for text analysis)

Tasks:

- identify relevant resources on http://www.jstatsoft.org/
- download HTML pages
- import them into R
- extract information via XPath
- (download PDFs)
- (import them into R)
- (extract information from PDFs)

How HTML forms work

- HTML forms are a collection of HTML nodes/tags: `<form>`, `<input>`, `<select>`, `<textfield>`
- forms collect input via text fields, radio buttons, checkmarks, etc.
- the elements describe the location of input elements as well as the HTTP method that is used to send information from client to server: GET/POST

Recall from HTML session

Form tag <form>

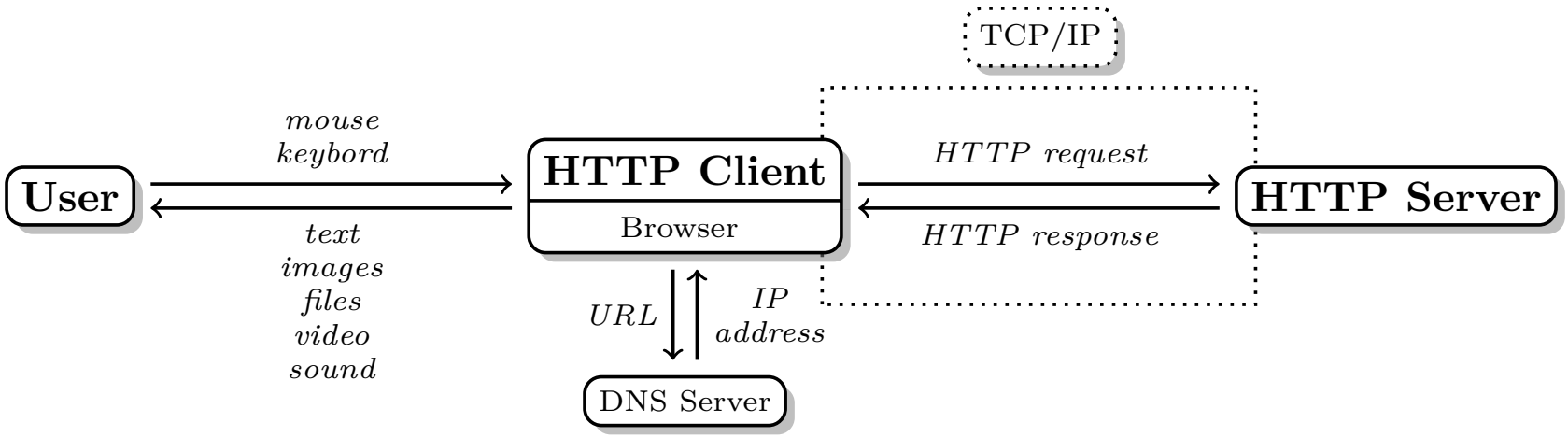
- allows to incorporate HTML forms
- client can send information to the HTTP server via forms

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>
```

A tiny bit of HTTP

Client-server communication with HTTP



A tiny bit of HTTP

Client-server communication with HTTP

1. Establishing connection

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

2. HTTP request

```
GET /index.html HTTP/1.1
Host: www.r-datacollection.com
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  ...

8  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
9  <html> <head>
10 <title></title>
11 </head>
12 ...
```

4. Closing connection

```
1  Closing connection #0
```

HTTP messages

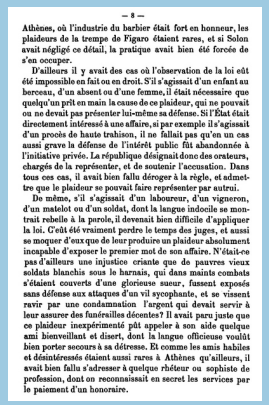
	schema	example
start line	`[method] [path] [version] [CRLF]`	`POST /greetings.html HTTP/1.1`
header	`[header name:] [header value] [CRLF]` `[CRLF]`	`Host: www.r-datacollection.com`
body	`[body]`	`Hi, there.` `How are you?`

	schema	example
start line	`[version] [status] [phrase] [CRLF]`	`HTTP/1.1 200 OK`
header	`[header name:] [header value] [CRLF]` `[CRLF]`	`Content-type: text/plain`
body	`[body]`	`I am fine, thank you very much.` `What else might I help you with?`

Case study: Checking texts on readability

Goal: upload texts and check them on readability
Tasks:

- inspect forms at http://read-able.com/
- upload texts via forms
- retrieve and process results



Exercises

1. The Wikipedia article at http://en.wikipedia.org/wiki/List_of_cognitive_biases provides several lists of various types of cognitive biases. Extract the information stored in the table on social biases. Each of the entries in the table points to another, more detailed article on Wikipedia. Fetch the list of references from each of these articles and store them in an adequate R object.

Part VIII

Good Practice

Overview

Good Practice

- Is web scraping legal?
- robots.txt
- Scraping etiquette

Is web scraping legal?

- no unambiguous yes or no in any country according to current jurisdiction
- so far, court cases (especially in the US) often (but not always) dealt with commercial interest and often (but not always) huge masses of data
  - eBay vs. Bidder's Edge
  - AP vs. Meltwater
  - Facebook vs. Pete Warden
  - United States vs. Aaron Swartz

A (not very useful) recommentation 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 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
5. consult current jurisdiction, e.g. on http://blawgsearch.justia.com or from a laywer specialized on internet law

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

Examples:

- Google
- NZZ

Syntax in *robots.txt*

Syntax

- not and 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

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

Separation of bots by empty line

```
User-agent: Googlebot
Disallow: /images/

User-agent: Slurp
Disallow: /images/
```

`Allow` declaration

```
User-agent: *
Disallow: /images/
Allow: /images/public/
```

Syntax in *robots.txt*

Crawl-delay (in seconds)

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

Robots `<meta>` tag

```
<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

Scraping etiquette