

# Automated Web Scrapping with R

Resul Umit

May 2021

[Skip intro — To the contents slide.](#)

# Who am I?

## Resul Umit

- post-doctoral researcher in political science at the University of Oslo
- teaching and studying representation, elections, and parliaments
  - **a recent publication**: Parliamentary communication allowances do not increase electoral turnout or incumbents' vote share
- teaching workshops, also on
  - **writing reproducible research papers**
  - **version control and collaboration**
  - **working with Twitter data**
  - **creating academic websites**
- more information available at **resulumit.com**

# The Workshop — Overview

- One day, on how to automate the process of extracting data from websites
  - 150+ slides, 15+ exercises
  - a **demonstration website** for practice
- Designed for researchers with basic knowledge of R programming language
  - does not cover programming with R
    - e.g., we will use existing functions and packages
  - ability to work with R will be very helpful
    - but not absolutely necessary — this ability can be developed during and after the workshop as well

# The Workshop — Motivation

- Data available on websites provide attractive opportunities for academic research
  - e.g., parliamentary websites were the main source of data for my PhD
- Acquiring such data requires
  - either a lot of resources, such as time
  - or a set of skills, such as automated web scraping
- Typically, such skills are not part of academic training
  - for my PhD, I hand-visited close to 3000 webpages to collect data manually
    - on members of ten parliaments
    - multiple times, to update the dataset as needed

# The Workshop — Motivation — Aims

- To provide you with an understanding of what is ethically possible
  - we will cover a large breath of issues, not all of it is for long-term memory
    - hence the slides are designed for self study as well
  - awareness of what is ethical and possible, Google, and perseverance are all you need
- To start you with acquiring and practicing the skills needed
  - practice with the demonstration website
    - plenty of data, stable structure, and an ethical playground
  - start working on a real project

# The Workshop — Contents

## Part 1. Getting the Tools Ready

- e.g., installing software

## Part 2. Preliminary Considerations

- e.g., ethics of web scraping

## Part 3. Webpage Source Code

- e.g., elements and selectors

## Part 4. Scraping Static Pages

- e.g., getting text from an element
- by using `rvest`

## Part 5. Scraping Dynamic Pages

- e.g., clicking before scraping
- by using `RSelenium` and `rvest`

[To the list of references.](#)

# The Workshop — Organisation

- Breakout in groups of two for exercises
  - participants learn as much from their partner as from instructors
  - switch partners after every other part
  - leave your breakout room manually, when everyone in the group is ready
- Type, rather than copy and paste, the code that you will find on these slides
  - typing is a part of the learning process
  - slides are, and will remain, available at [resulimit.com/teaching/scrp\\_workshop.html](https://resulimit.com/teaching/scrp_workshop.html)
- When you have a question
  - ask your partner
  - google together
  - ask me

# The Workshop — Organisation — Slides

03 : 00

Slides with this background colour indicate that your action is required, for

- setting the workshop up
  - e.g., installing R
- completing the exercises
  - e.g., checking website protocols
  - these slides have countdown timers
    - as a guide, not to be followed strictly



# The Workshop — Organisation — Slides

- Code and text that go in R console or scripts appear as such – in a different font, on gray background
  - long codes and texts will have their own line(s)

```
bow(url = "https://luzpar.netlify.app/members/") %>%  
  scrape() %>%  
  html_elements(css = "td+ td a") %>%  
  html_attr("href") %>%  
  url_absolute(base = "https://luzpar.netlify.app/")
```

# The Workshop — Organisation — Slides

- Codes and texts that go in R console or scripts appear as such – in a different font, on gray background
  - long codes and texts will have their own line(s)
- Results that come out as output appear as such — in the same font, on green background
  - with some results, such as browsers popping up
- Specific sections are highlighted yellow as such for emphasis
  - these could be for anything — codes and texts in input, results in output, and/or texts on slides
- The slides are designed for self-study as much as for the workshop
  - *accessible*, in substance and form, to go through on your own

# Part 1. Getting the Tools Ready

[Back to the contents slide.](#)

# Workshop Slides — Access on Your Browser

- Having the workshop slides\* on your own machine might be helpful
  - flexibility to go back and forward on your own
    - especially while in a breakout room
  - ability to scroll across long codes on some slides
- Access at [https://resulunit.com/teaching/scrp\\_workshop.html](https://resulunit.com/teaching/scrp_workshop.html)
  - will remain accessible after the workshop
  - might crash for some Safari users
    - if using a different browser application is not an option, view the [PDF version of the slides](#) on GitHub

\* These slides are produced in R, with the `xaringan` package ([Xie, 2020](#)).

# Demonstration Website — Explore on Your Browser

05 : 00

- There is a demonstration website for this workshop
  - available at <https://luzpar.netlify.app/>
  - includes fabricated data on the imaginary Parliament of Luzland
  - provides us with plenty of data, stable structure, and an ethical playground
- Using this demonstration website for practice is recommended
  - tailored to exercises, no ethical concern
  - but not compulsory — use a different one if you prefer so
- Explore the website now
  - click on the links to see an individual page for
    - states, constituencies, members, and documents
  - notice that the documents section is different than the rest
    - it is a page with dynamic frame

# R — Download from the Internet and Install

- Programming language of this workshop
  - created for data analysis, extending for other purposes
    - e.g., accessing websites
  - allows for all three steps in one environment
    - accessing websites; scraping and processing data
  - an alternative: `python`
- Download R from <https://cloud.r-project.org>
  - optional, if you have it already installed — but then consider updating\*
    - the `R.version.string` command checks the version of your copy
    - compare with the latest official release at <https://cran.r-project.org/sources.html>

\* The same applies to all software that follows — consider updating if you have them already installed. This ensures everyone works with the latest, exactly the same, tools.

# RStudio — Download from the Internet and Install

- Optional, but highly recommended
  - facilitates working with R
- A popular integrated development environment (IDE) for R
  - an alternative: **GNU Emacs**
- Download RStudio from <https://rstudio.com/products/rstudio/download>
  - choose the free version
  - to check for any updates, follow from the RStudio menu:

Help -> Check for Updates

# RStudio Project — Create from within RStudio

- RStudio allows for dividing your work with R into separate projects
  - each project gets dedicated workspace, history, and source documents
  - [this page](#) has more information on why projects are recommended
- Create a new RStudio project for for this workshop, following from the RStudio menu:
  - File -> New Project -> New Directory -> New Project
- Choose a location for the project with Browse . . .
  - avoid choosing a synced location, e.g., Dropbox
    - likely to cause warning and/or error messages
    - if you must, pause syncing, or add an sync exclusion



# R Packages — Install from within RStudio\*

02:00

Install the packages that we need

```
install.packages(c("rvest", "RSelenium", "robotstxt", "polite", "dplyr"))
```

\* You may already have a copy of one or more of these packages. In that case, I recommend updating by re-installing them now.

# R Packages — Install from within RStudio

Install the packages that we need

```
install.packages(c("rvest", "RSelenium", "robotstxt", "polite", "dplyr"))
```

We will use

- `rvest` (Wickham, 2021), for scraping websites
- `RSelenium` (Harrison, 2020), for browsing the web programmatically
- `robotstxt` (Meissner & Ren, 2020), for checking permissions to scrape websites
- `polite` (Perepolkin, 2019), for compliance with permissions to scrape websites
- `dplyr` (Wickham et al, 2021), for data manipulation

# R Script — Start Your Script

- Check that you are in the newly created project
  - indicated at the upper-right corner of RStudio window
- Create a new R Script, following from the RStudio menu

File -> New File -> R Script

- Name and save your file
  - to avoid the Untitled123 problem
  - e.g., scrape\_web.R
- Load rvest and other packages

```
library(rvest)
library(RSelenium)
library(robotstxt)
library(polite)
library(dplyr)
```

# Java — Download from the Internet and Install

- A language and software that RSelenium needs
  - for automation scripts
- Download Java from <https://www.java.com/en/download/>
  - requires restarting any browser that you might have open

# Chrome — Download from the Internet and Install

- A browser that facilitates web scraping
  - favoured by R Selenium and most programmers
- Download Chrome from <https://www.google.com/chrome/>

# SelectorGadget — Add Extension to Browser

- An extension for Chrome
  - facilitates selecting what to scrape from a webpage
  - optional, but highly recommended
  - open source software
- Add the extension to your browser
  - search for it at <https://chrome.google.com/webstore/category/extensions>
  - if you cannot use Chrome, drag and drop this link to your bookmarks bar
- ScrapeMate is an alternative extension
  - for both Chrome and Firefox
  - on Firefox, search at <https://addons.mozilla.org/>

# Other Resources\*

- R Selenium vignettes
  - available at <https://cran.r-project.org/web/packages/R Selenium/vignettes/basics.html>
- R for Data Science (Wickham & Grolemund, 2019)
  - open access at <https://r4ds.had.co.nz>
- Text Mining with R: A Tidy Approach (Silge & Robinson, 2017)
  - open access at [tidytextmining.com](https://tidytextmining.com)
  - comes with a [course website](#) where you can practice

\* I recommend these to be consulted not during but after the workshop.

## Part 2. Preliminary Considerations

[Back to the contents slide.](#)



# Considerations — the Law

- Web scraping might be illegal
  - depending on who is scraping what, why, how — and under which jurisdiction
  - reflect, and check, before you scrape
- Web scraping might be more likely to be illegal if, for example,
  - it is harmful to the source
    - commercially
      - e.g., scraping a commercial website to create a rival website
    - physically
      - e.g., scraping a website so hard and fast that it collapses
  - it gathers data that is
    - under copyright
    - not meant for the public to see
    - then used for financial gain

# Considerations — the Ethics

- Web scraping might be unethical
  - depending on who is scraping what, why, and how
  - reflect before you scrape
- Web scraping might be more likely to be unethical if, for example,
  - it is — edging towards — being illegal
  - it does not respect the restrictions
    - as defined in `robots.txt` files
  - it harvests data
    - that is otherwise available to download, e.g., through APIs
    - without purpose, at dangerous speed, repeatedly

# Considerations — the Ethics — robots.txt

- Most websites declare a robots exclusion protocol
  - making their rules known with respect to programmatic access
    - who is (not) allowed to scrape what, and sometimes, at what speed
  - within robots.txt files
    - available at, e.g., [www.websiteurl.com/robots.txt](http://www.websiteurl.com/robots.txt)
- The rules in robots.txt cannot not enforced upon scrapers
  - but should be respected for ethical reasons
- The language in robots.txt files is specific but intuitive
  - easy to read and understand
  - the robotstxt package makes these even easier

# Considerations — the Ethics — robots.txt — Syntax

- It has pre-defined keys, most importantly
  - User-agent indicates who the protocol is for
  - Allow indicates which part(s) of the website can be scraped
  - Disallow indicates which part(s) must not be scraped
  - Crawl-delay indicates how fast the website could be scraped
- In case you write your own protocol one day, note that
  - the keys start with capital letters
  - they are followed by a colon :

```
User-agent:  
Allow:  
Disallow:  
Crawl-delay:
```

# Considerations — the Ethics — robots.txt — Syntax

- Websites define their own values
  - after a colon and a white space
- Note that
  - \* indicates the protocol is for everyone
  - / indicates all sections and pages
  - /about/ indicates a specific path
  - values for Crawl-delay are in seconds
  - this website allows anyone to scrape, provided that
    - /about/ is left out, and
    - the website is accessed at 5-seconds intervals

```
User-agent: *  
Allow: /  
Disallow: /about/  
Crawl-delay: 5
```

# Considerations — the Ethics — robots.txt — Syntax

Files might include optional comments, written after the number sign #

```
# thank you for respecting our protocol
```

```
User-agent: *
```

```
Allow: /
```

```
Disallow: /about/
```

```
Crawl-delay: 5 # five second delay, to ensure our servers are not overloaded
```

# Considerations — the Ethics — robots.txt — Syntax

The protocol of this website only applies to Google

- Google is allowed to scrape everything
- there is no defined rule for anyone else

```
User-agent: googlebot  
Allow: /
```

# Considerations — the Ethics — robots.txt — Syntax

The protocol of this website only applies to Google

- Google is **disallowed** to scrape **two** specific paths
  - with no limit on speed
- there is no defined rule for anyone else

```
User-agent: googlebot  
Disallow: /about/  
Disallow: /history/
```



# Considerations — the Ethics — robots.txt — Syntax

This website has different protocols for different agents

- Google is allowed to scrape everything, with a 5-second delay
- Bing is not allowed to scrape anything
- everyone else can scrape the section or page located at [www.websiteurl/about/](http://www.websiteurl/about/)

```
User-agent: googlebot
```

```
Allow: /
```

```
Crawl-delay: 5
```

```
User-agent: bing
```

```
Disallow: /
```

```
User-agent: *
```

```
Allow: /about/
```

# Considerations — the Ethics — robots.txt

- The robots.txt package facilitates checking website protocols
  - from within R — no need to visit websites via browser
  - provides functions to check, among others, the rules for specific paths and/or agents
- There are two main functions
  - robots.txt, which gets complete protocols
  - paths\_allowed, which checks protocols for one or more specific paths

# Considerations — the Ethics — robots.txt

Use the `robots.txt` function to get a protocol

- supply a base url with the `domain` argument
  - as a string
  - probably the only argument that you will need

```
robots.txt(  
    domain = NULL,  
    ...  
)
```

# Considerations — the Ethics — robots.txt

```
robotstxt(domain = "https://luzpar.netlify.app")
```

```
## $domain
## [1] "https://luzpar.netlify.app"
##
## $text
## [robots.txt]
## -----
##
## User-agent: googlebot
## Disallow: /states/
##
## User-agent: *
## Allow: /
## Crawl-delay: 2
##
##
##
##
## $robexclobj
```

# Considerations — the Ethics — robots.txt

Use the `paths_allowed` function to check protocols for one or more specific paths

- supply a base url with the `domain` argument
- `path` and `bot` are the other important arguments
  - notice the default values
- leads to either `TRUE` (allowed to scrape) or `FALSE` (not allowed)

```
paths_allowed(  
  domain = "auto",  
  paths = "/",  
  bot = "*",  
  ...  
)
```

# Considerations — the Ethics — robots.txt

```
paths_allowed(domain = "https://luzpar.netlify.app")
```

```
## [1] TRUE
```

```
paths_allowed(domain = "https://luzpar.netlify.app",  
              paths = c("/states/", "/constituencies/"))
```

```
## [1] TRUE TRUE
```

```
paths_allowed(domain = "https://luzpar.netlify.app",  
              paths = c("/states/", "/constituencies/"), bot = "googlebot")
```

```
## [1] FALSE TRUE
```

# Exercises

07:30

1) Check the protocols for <https://www.theguardian.com>

- via a browser and with the `robotstxt` function
- compare what you see

2) Check a path with the `paths_allowed` function

- such that it will return `FALSE`
- taking the information from Exercise 1 into account

3) Check the protocols for any website that you might wish to scrape

- with the `robotstxt` function

# Considerations — the Ethics — Speed

- Websites are designed for visitors with human-speed in mind
  - computer-speed visits can overload servers, depending on bandwidth
    - popular websites might have more bandwidth
    - but, they might attract multiple scrapers at the same time
- Waiting a little between two visits makes scraping more ethical
  - waiting time may or may not be defined in the protocol
    - lookout for, and respect, the `Crawl-delay` key in `robots.txt`
  - [Part 4](#) and [Part 5](#) covers how to wait
- Not waiting enough might lead to a ban
  - by site owners, administrators
  - for IP addresses with undesirably high number of visits in a short period of time



# Considerations — the Ethics — Purpose

Ideally, we scrape for a purpose

- e.g., for academics, to answer one or more research questions, test hypotheses
  - developed prior to data collection, analysis
    - based on, e.g., theory, claims, observations
  - perhaps, even pre-registered
    - e.g., at [OSF Registries](#)

# Considerations — Data Storage

Scraped data frequently requires

- large amounts of digital storage space
  - internet data is typically big data
- private, safe storage spaces
  - due to local rules, institutional requirements

# Part 3. Webpage Source Code

[Back to the contents slide.](#)

# Webpage Source Code — Overview

- Webpages include more than what is immediately visible to visitors
  - not only text, images, links
  - but also code for structure, style, and functionality — interpreted by browsers first
    - HTML provides the structure
    - CSS provides the style
    - JavaScript provides functionality, if any
- Web scraping requires working with the source code
  - even when scraping only what is already visible
  - to choose one or more desired parts of the visible
    - e.g., text in table and/or bold only
- Source code also offers more, invisible, data to be scraped
  - e.g., URLs hidden under text

# Webpage Source Code — View in Browser

The `Ctrl + U` shortcut is to display source code — alternatively, right click and `View Page Source`

# Webpage Source Code — View in Browser — DOM

Browsers also offer putting source codes in a structure

- known as DOM (document object model), initiated by the F12 key on Chrome

# Exercises

05:00

## 4) View the source code of a page

- as plain code and as in DOM
- compare the look of the two

## 5) Search for a word or a phrase in source code

- copy from the front-end page
- search in plain text code or in DOM
  - using the `Ctrl + F` shortcut
- compare the look of the front- and back-end

# Webpage Source Code — HTML — Overview

- HTML stands for **hypertext markup language**
  - it gives the structure to what is visible to visitors
    - text, images, links
  - would a piece of text appear in a paragraph or a list?
    - depends on the HTML code around that text

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```



# Webpage Source Code — HTML — Overview

## HTML documents

- start with a declaration
  - so that browsers know what they are

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — HTML — Overview

## HTML documents

- start with a declaration
  - so that browsers know what they are
- consist of elements
  - written in between opening and closing tags

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — HTML — the Root

`html` is the root element

- it is also the parent to all other elements
- its important children are the head and body elements

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — HTML — the Head

**head** contains metadata, such as

- titles, which appear in browser bars and tabs
- style elements

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — HTML — Overview

**body** contains the elements in the main body of pages, such as

- headers, paragraphs, lists, tables, images

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — HTML — Syntax

Most elements have opening and closing tags

```
<p>This is a one sentence paragraph.</p>
```

This is a one sentence paragraph.

Note that

- tag name, in this case **p**, defines the structure of the element
- the closing tag has a forward slash **/** before the element name

# Webpage Source Code — HTML — Syntax

Most elements have some content

```
<p>This is a one sentence paragraph.</p>
```

This is a one sentence paragraph.

# Webpage Source Code — HTML — Syntax

Elements can be nested

```
<p>This is a <strong>one</strong> sentence paragraph.</p>
```

This is a **one** sentence paragraph.

Note that

- there are two elements above, a paragraph and a strong emphasis
- strong is said to be the child of the paragraph element
  - there could be more than one child
  - in that case, children are numbered from the left
- paragraph is said to be the parent of the strong element



# Webpage Source Code — HTML — Syntax

Elements can have **attributes**

```
<p>This is a <strong id="sentence-count">one</strong> sentence paragraph.</p>
```

This is a **one** sentence paragraph.

Note that

- the id attribute is not visible to the visitors
- attribute string **sentence-count** could have been anything I could come up with
  - unlike the tag and attribute names — e.g., strong, id as they are pre-defined
- there are some other attributes that are visible

# Webpage Source Code — HTML — Syntax

There could be more than one attribute in an element

```
<p>This is a <strong class="count" id="sentence-count">one</strong> sentence paragraph.</p>  
<p>There are now <strong class="count" id="paragraph-count">two</strong> paragraphs.</p>
```

This is a **one** sentence paragraph.

There are now **two** paragraphs.

Note that

- there could be more than one attribute in an element
  - with a white space in between them
- the `class` attribute can apply to multiple elements
  - while the `id` attribute must be unique on a page

# Webpage Source Code — HTML — Important Elements & Attributes

Links are provided with the a (anchor) element

```
<p>Click <a href="https://www.google.com/">here</a> to google things.</p>
```

Click [here](https://www.google.com/) to google things.

Note that

- href (hypertext reference) is a required attribute for this element
- most attributes are optional, some are required

# Webpage Source Code — HTML — Links

## Links

```
<p>Click <a title="This text appears when visitors hover over the link"  
href="https://www.google.com/">here</a> to google things.</p>
```

Click [here](https://www.google.com/) to google things.

Note that

- the `title` attribute is one of the optional attributes
- it becomes visible when hovered over with mouse

# Webpage Source Code — HTML — Lists

The `<ul>` tag introduces unordered lists, while the `<li>` tag defines lists items

```
<ul>
  <li>books</li>
  <li>journal articles</li>
  <li>reports</li>
</ul>
```

- books
- journal articles
- reports

Note that

- Ordered lists are introduced with the `<ol>` tag instead

# Webpage Source Code — HTML — Notes

By default, multiple spaces and/or lines breaks are ignored by browsers

```
<ul><li>books</li><li>journal          articles</li><li>reports
</li>

</ul>
```

- books
- journal articles
- reports

Note that

- plain source code may or may not be written in a readable manner
- this is one reason why DOM is helpful

# Exercises

10:00

6) Re-create the **body** of the page at <https://luzpar.netlify.app/states/> in R

- start an HTML file, following from the RStudio menu:

```
File -> New File -> HTML File
```

- copy the text from the website, paste in the HTML file
- add the structure with HTML code
- click Preview to view the result

7) Add an attribute to an element

- which is not already on the original page
- hints:
  - there are at least two attributes for the list elements on the original page
  - google to see what attributes list items accept
  - <https://www.w3schools.com/> is a great place to look at
  - save this document as we will continue working on it

# Webpage Source Code — CSS — Overview

- CSS stands for **cascading stylesheets**
  - it gives the style to what is visible to visitors
    - text, images, links
  - would a piece of text appear in black or blue?
    - depends on the CSS for that text
- CSS can be defined
  - inline, as an attribute of an element
  - internally, as a child element of the head element
  - externally, but then linked in the head element



# Webpage Source Code — CSS — Syntax

CSS is written in rules

```
p {font-size:12px;}
```

```
.count {background-color:yellow;}
```

```
#sentence-count {color:red;}
```

# Webpage Source Code — CSS — Syntax

- CSS is written in rules, with a syntax consisting of
  - one or more **selectors**, matching one or more HTML elements and/or attributes
- Note that
  - the syntax changes with the selector type
    - elements and attributes
    - among attributes, classes and ids

```
p {font-size:14px;}  
h1 h2 {color:blue;}  
.count {background-color:yellow;}  
#sentence-count {color:red; font-size:14px;}
```

# Webpage Source Code — CSS — Syntax

- CSS is written in rules, with a syntax consisting of
  - one or more selectors, matching one or more HTML elements and/or attributes
  - a **declaration**
- Note that
  - declarations are written in between two curly brackets

```
p {font-size:14px;}  
h1 h2 {color:blue;}  
.count {background-color:yellow;}  
#sentence-count {color:red; font-size:14px;}
```

# Webpage Source Code — CSS — Syntax

- CSS is written in rules, with a syntax consisting of
  - one or more selectors, matching one or more HTML elements and/or attributes
  - a declaration, with one or more **properties**
- Note that
  - properties are followed by a colon

```
p {font-size:14px;}  
h1 h2 {color:blue;}  
.count {background-color:yellow;}  
#sentence-count {color:red; font-size:14px;}
```

# Webpage Source Code — CSS — Syntax

- CSS is written in rules, with a syntax consisting of
  - one or more selectors
  - a declaration, with one or more properties and values
- Note that
  - values are followed by a semicolon
  - property:value; pairs are separated by a white space

```
p {font-size:14px;}  
h1 h2 {color:blue;}  
.count {background-color:yellow;}  
#sentence-count {color:red; font-size:14px;}
```

# Webpage Source Code — CSS — Internal

- CSS rules can be defined internally
  - within the `style` element
  - as a child of the `head` element
- Internally defined rules apply to all matching selectors
  - on the same page

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      h1 {color: blue;}
    </style>
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — CSS — External

- CSS rules can be defined externally
  - saved somewhere linkable
  - defined with the the linked element
  - as a child of the head element
- Externally defined rules
  - are saved in a file with .css extension
  - apply to all matching selectors
    - on any page linked

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="styles" href="simple.css">
    <title>A title for browsers</title>
  </head>
  <body>
    <h1>A header</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>This</li>
      <li>is a</li>
      <li>list</li>
    </ul>
  </body>
</html>
```

# Webpage Source Code — CSS — Inline

CSS rules can also be defined inline

- with the `style` attribute
- does not require selector
- applies only to that element

```
<p>This is a <strong style="color:blue;">one</strong> sentence paragraph.</p>
```

This is a **one** sentence paragraph.



# Exercise

05:00

8) Provide some simple style to your HTML document

- one that you created during the previous exercise
- using internal or external style, but not inline
  - so that you can practice selecting element and/or attributes
- no idea what to do? try
  - increasing the font size of the text in paragraph
  - change the colour of the second item in the list to red

# Part 3. Scraping Static Pages

[Back to the contents slide.](#)

# Static Pages — Overview

- We will collect data from static pages with the `rvest` package
  - static pages are those that display the same source code to all visitors
    - including the content — it does not change
  - every visitor sees the same page at a given URL
  - each page has a different URL
  - <https://luzpar.netlify.app/> is a static page
- Scraping static pages involves two main tasks
  - download the source code from one or more pages to R
    - typically, the only interaction with the page itself
  - select the exact information needed from the source code
    - takes place locally, on your machine
    - the main functionality that `rvest` offers
      - with the help from selectors

# Static Pages — `rvest` — Overview

- A relative small R package for web scraping
  - created by [Hadley Wickham](#)
  - popular — used by many for web scraping
    - downloaded 622,724 times last month
    - some of it must be thanks to being a part of the tidyverse family
  - last major revision was in March 2021
    - better alignment with tidyverse
- A lot has already been written on this package
  - you will find solutions to, or help for, any issues online
  - see first the [package documentation](#), numerous tutorials — such as [this](#) and [this](#), and [this](#)
- Comes with the recommendation to combine it with the `polite` package
  - for ethical web scraping

# Static Pages — `rvest` — Get Source Code

Use the `read_html` function to get the source code of a webpage into R

```
read_html("https://luzpar.netlify.app/")
```

```
## {html_document}  
## <html lang="en-us">  
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...  
## [2] <body id="top" data-spy="scroll" data-offset="70" data-target="#navb ...
```

# Static Pages — rvest — Get Source Code

You may wish to check the protocol first

```
paths_allowed(domain = "https://luzpar.netlify.app/")
```

```
## [1] TRUE
```

```
read_html("https://luzpar.netlify.app/")
```

```
## {html_document}
## <html lang="en-us">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
## [2] <body id="top" data-spy="scroll" data-offset="70" data-target="#navb ...
```

# Static Pages — `rvest` — Get Source Code — `polite`

- The `polite` package facilitates ethical scraping
  - recommended by `rvest`
- It divides the process of getting source code into two
  - check the protocol
  - get the source only if allowed
- It also
  - waits for a period of time
    - minimum by what is specified in the protocol
  - allows you to introduce yourself to website administrators while scraping

# Static Pages — `rvest` — Get Source Code — `polite`

- First, use the `bow` function to check the protocol
  - for a specific `URL`

```
bow(url,  
    user_agent = "polite R package - https://gi  
    delay = 5,  
    ...  
)
```



# Static Pages — `rvest` — Get Source Code — `polite`

- First, use the `bow` function to check the protocol
  - for a specific URL
  - for a specific `agent`
- Note that
  - the `user_agent` argument can communicate information to website administrators
    - e.g., your name and contact details

```
bow(url,  
    user_agent = "polite R package - https://gi  
    delay = 5,  
    force = FALSE,  
    ...  
)
```

# Static Pages — rvest — Get Source Code — polite

- First, use the bow function to check the protocol

- for a specific URL
- for a specific agent
- for any **crawl-delay directives**

```
bow(url,  
    user_agent = "polite R package - https://gi  
    delay = 5,  
    force = FALSE,  
    ...  
)
```

- Note that
  - the delay argument cannot be set to a number smaller than in the directive
    - if there is one

# Static Pages — rvest — Get Source Code — polite

- First, use the bow function to check the protocol

- for a specific URL
- for a specific agent
- for crawl-delay directives

```
bow(url,  
    user_agent = "polite R package - https://gi  
    delay = 5,  
    force = FALSE,  
    ...  
)
```

- Note that

- the delay argument cannot be set to a number smaller than in the directive
  - if there is one
- the force argument is set to FALSE by default
  - avoids repeated, unnecessary interactions with web page
  - by caching, and re-using, previously downloaded sources

# Static Pages — rvest — Get Source Code — polite

- Second, use the `scrape` function get source code
  - for an object created with the `bow` function
- Note that
  - `scrape` will only work if the results from `bow` are positive
    - creating a safety valve for ethical scraping
  - by piping, `bow` into `scrape`, you can avoid creating objects

```
scrape(bow,  
      ...  
      )
```

# Static Pages — rvest — Get Source Code

These two are equal, when there is no protocol against the access

```
read_html("https://luzpar.netlify.app/")
```

```
## {html_document}
## <html lang="en-us">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
## [2] <body id="top" data-spy="scroll" data-offset="70" data-target="#navb ...
```

```
bow(url = "https://luzpar.netlify.app/") %>%
  scrape()
```

```
## {html_document}
## <html lang="en-us">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset= ...
## [2] <body id="top" data-spy="scroll" data-offset="70" data-target="#navb ...
```

# Static Pages — rvest — Get Source Code

The difference occurs when there is a protocol against the access

```
bow(url = "https://luzpar.netlify.app/states/", user_agent = "googlebot")
```

```
## <polite session> https://luzpar.netlify.app/states/  
##   User-agent: googlebot  
##   robots.txt: 2 rules are defined for 2 bots  
##   Crawl delay: 5 sec  
##   The path is not scrapable for this user-agent
```

```
bow(url = "https://luzpar.netlify.app/states/", user_agent = "googlebot") %>%  
  scrape()
```

```
## Warning: No scraping allowed here!  
  
## NULL
```

# Static Pages — rvest — html\_elements

- Get one or more HTML elements
  - specified with a selector
    - css or xpath
  - we will work with css in this workshop
    - facilitated by Chrome and SelectorGadget
- Note that
  - there are two versions of the same function
    - singular one gets the first instance of an element
    - plural one gets all instances

```
html_element(x,  
             css,  
             xpath)  
  
html_elements(x,  
             css,  
             xpath)
```

# Static Pages — rvest — html\_elements

Get the anchor (a) elements on the homepage

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "a")
```

```
## {xml_nodeset (24)}  
## [1] <a class="js-search" href="#" aria-label="Close"><i class="fas fa-t ...  
## [2] <a class="navbar-brand" href="/">Parliament of Luzland</a>  
## [3] <a class="navbar-brand" href="/">Parliament of Luzland</a>  
## [4] <a class="nav-link active" href="/"><span>Home</span></a>  
## [5] <a class="nav-link" href="/states/"><span>States</span></a>  
## [6] <a class="nav-link" href="/constituencies/"><span>Constituencies</s ...  
## [7] <a class="nav-link" href="/members/"><span>Members</span></a>  
## [8] <a class="nav-link" href="/documents/"><span>Documents</span></a>  
## [9] <a class="nav-link js-search" href="#" aria-label="Search"><i class ...  
## [10] <a href="#" class="nav-link" data-toggle="dropdown" aria-haspopup=" ...  
## [11] <a href="#" class="dropdown-item js-set-theme-light"><span>Light</s ...  
## [12] <a href="#" class="dropdown-item js-set-theme-dark"><span>Dark</spa ...  
## [13] <a href="#" class="dropdown-item js-set-theme-auto"><span>Automatic ...  
## [14] <a href="https://github.com/resulunit/scrp_workshop" target="_blank ...  
## [15] <a href="https://resulunit.com/" target="_blank" rel="noopener">Res ...
```



# Static Pages — rvest — html\_elements

I would like to get only the URLs on the main body of the page

- SelectorGadget tells me the correct selector is #title a

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "#title a")
```

```
## {xml_nodeset (9)}  
## [1] <a href="https://github.com/resulimit/scrp_workshop" target="_blank" ...  
## [2] <a href="https://resulimit.com/" target="_blank" rel="noopener">Resu ...  
## [3] <a href="/documents/">documents</a>  
## [4] <a href="/constituencies/">constituencies</a>  
## [5] <a href="/members/">members</a>  
## [6] <a href="/states/">states</a>  
## [7] <a href="https://github.com/rstudio/blogdown" target="_blank" rel="n ...  
## [8] <a href="https://gohugo.io/" target="_blank" rel="noopener">Hugo</a>  
## [9] <a href="https://github.com/wowchemy" target="_blank" rel="noopener" ...
```

# Static Pages — `rvest` — `html_text`

- Get the text content of one or more HTML elements
  - for the elements already chosen
    - with the `html_elements` function
  - this returns what is already visible to visitors
- Note that
  - there are two versions of the same function
    - `html_text` returns text with any space or line breaks around it
    - `html_text2` returns plain text

```
html_text(x, trim = FALSE)
```

```
html_text2(x, preserve_nbsp = FALSE)
```

# Static Pages — rvest — html\_text

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "#title a") %>%  
  html_text()
```

```
## [1] "a workshop on automated web scraping"  
## [2] "Resul Umit"  
## [3] "documents"  
## [4] "constituencies"  
## [5] "members"  
## [6] "states"  
## [7] "Blogdown"  
## [8] "Hugo"  
## [9] "Wowchemy"
```

# Static Pages — rvest — html\_attr

- Get one or more attributes of one or more HTML elements
  - for the elements already chosen
    - with the `html_elements` function
  - attributes are specified with their name
    - not css or xpath
- Note that
  - there are two versions of the same function
    - singular one gets a specified attribute
    - plural one gets all available attributes

```
html_attr(x, name, default = NA_character_)
```

```
html_attrs(x)
```

# Static Pages — rvest — html\_attrs

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "#title a") %>%  
  html_attrs()
```

```
## [[1]]  
##                                href  
## "https://github.com/resulumit/scrp_workshop"  
##                                target  
##                                "_blank"  
##                                rel  
##                                "noopener"  
##  
## [[2]]  
##                                href                                target                                rel  
## "https://resulumit.com/"                                "_blank"                                "noopener"  
##  
## [[3]]  
##                                href  
## "/documents/"  
##
```

# Static Pages — rvest — html\_attr

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "#title a") %>%  
  html_attr(name = "href")
```

```
## [1] "https://github.com/resulunit/scrp_workshop"  
## [2] "https://resulunit.com/"  
## [3] "/documents/"  
## [4] "/constituencies/"  
## [5] "/members/"  
## [6] "/states/"  
## [7] "https://github.com/rstudio/blogdown"  
## [8] "https://gohugo.io/"  
## [9] "https://github.com/wowchemy"
```

Note that

- some URLs are given relative to the base URL
  - e.g., /states/, which is actually /states/
  - you can complete them with the `url_absolute` function

# Static Pages — rvest — url\_absolute

Complete the relative URLs with the url\_absolute function

```
bow(url = "https://luzpar.netlify.app") %>%  
  scrape() %>%  
  html_elements(css = "#title a") %>%  
  html_attr(name = "href") %>%  
  url_absolute(base = "https://luzpar.netlify.app")
```

```
## [1] "https://github.com/resulmit/scrp_workshop"  
## [2] "https://resulmit.com/"  
## [3] "https://luzpar.netlify.app/documents/"  
## [4] "https://luzpar.netlify.app/constituencies/"  
## [5] "https://luzpar.netlify.app/members/"  
## [6] "https://luzpar.netlify.app/states/"  
## [7] "https://github.com/rstudio/blogdown"  
## [8] "https://gohugo.io/"  
## [9] "https://github.com/wowchemy"
```

# Static Pages — rvest — html\_table

Use the `html_table()` function to get the text content of table elements

```
bow(url = "https://luzpar.netlify.app/members/") %>%  
  scrape() %>%  
  html_elements(css = "table") %>%  
  html_table()
```

```
## [[1]]  
## # A tibble: 100 x 3  
##   Member      Constituency Party  
##   <chr>      <chr>      <chr>  
## 1 Arthur Ali    Mühlshafen  Liberal  
## 2 Chris Antony  Benwerder   Labour  
## 3 Chloë Bakker  Steffisfelden Labour  
## 4 Rose Barnes   Dillon      Liberal  
## 5 Emilia Bauer  Kilnard     Green  
## 6 Wilma Baumann Granderry    Green  
## 7 Matteo Becker Enkmelo     Labour  
## 8 Patricia Bernard Gänsernten  Labour  
## 9 Lina Booth    Leonrau     Liberal
```



# Static Pages — rvest

We can create the same tibble with `html_text`, which requires getting each variable separately to be merged

```
tibble(  
  "Member" = bow(url = "https://luzpar.netlify.app/members/") %>%  
    scrape() %>%  
    html_elements(css = "td:nth-child(1)") %>%  
    html_text(),  
  "Constituency" = bow(url = "https://luzpar.netlify.app/members/") %>%  
    scrape() %>%  
    html_elements(css = "td:nth-child(2)") %>%  
    html_text(),  
  "Party" = bow(url = "https://luzpar.netlify.app/members/") %>%  
    scrape() %>%  
    html_elements(css = "td:nth-child(3)") %>%  
    html_text()  
)
```

# Static Pages — rvest

Keep the number of interactions with websites to minimum

- by saving the source code as an object, which could be used repeatedly

```
the_page <- bow(url = "https://luzpar.netlify.app/members/") %>%  
  scrape()  
  
tibble(  
  "Member" = the_page %>%  
    html_elements(css = "td:nth-child(1)") %>%  
    html_text(),  
  "Constituency" = the_page %>%  
    html_elements(css = "td:nth-child(2)") %>%  
    html_text(),  
  "Party" = the_page %>%  
    html_elements(css = "td:nth-child(3)") %>%  
    html_text()  
)
```

# Exercise

15:00

9) Create a dataframe out of the table at <https://luzpar.netlify.app/members/>

- with as many variables as possible
- hints:
  - start with the code in the previous slide, and add new variables from attributes
  - the first two columns have important attributes
    - e.g., URLs for the pages for members and their constituencies
    - make these URLs absolute
    - see what other attributes are there to collect

# Static Pages — Crawling — Overview

- Rarely a single page includes all variables that we need
  - instead, they are often scattered across different pages of a website
  - e.g., we might need data on election results — in addition to constituency names
- Web scraping then requires crawling across pages
  - using information found on one page, to go to the next
  - website design may or may not facilitate crawling
- We can write for loops to crawl
  - the speed of our code matters the most when we crawl
  - ethical concerns are higher

# Static Pages — Crawling — Example

## Task:

- I need data on the name and vote share of parties that came second in each constituency
- This data is available on constituency pages, but
  - there are too many such pages
    - e.g., <https://luzpar.netlify.app/constituencies/arford/>
  - I do not have the URL to these pages

## Plan:

- Scrape <https://luzpar.netlify.app/members/> for URLs
- Write a for loop to
  - visit these pages one by one
  - collect and save the variables needed
  - write these variables into a list
  - turn the list into a dataframe

# Static Pages — Crawling — Example

Scrape the page that has all URLs, for absolute URLs

```
the_links <- bow(url = "https://luzpar.netlify.app/members/") %>%  
  scrape() %>%  
  html_elements(css = "td+ td a") %>%  
  html_attr("href") %>%  
  url_absolute(base = "https://luzpar.netlify.app/")  
  
# check if it worked  
head(the_links)
```

```
## [1] "https://luzpar.netlify.app/constituencies/muhlshafen/"  
## [2] "https://luzpar.netlify.app/constituencies/benwerder/"  
## [3] "https://luzpar.netlify.app/constituencies/steffisfelden/"  
## [4] "https://luzpar.netlify.app/constituencies/dillon/"  
## [5] "https://luzpar.netlify.app/constituencies/kilnard/"  
## [6] "https://luzpar.netlify.app/constituencies/granderry/"
```

# Static Pages — Crawling — Example

Create an empty list

```
temp_list <- list()

for (i in 1:length(the_links)) {
  the_page <- bow(the_links[i]) %>% scrape()
  temp_tibble <- tibble(
    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),
    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),
    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()
  )
  temp_list[[i]] <- temp_tibble
}

df <- as_tibble(do.call(rbind, temp_list))
```

# Static Pages — Crawling — Example

Start a for loop to iterate over the links one by one

```
temp_list <- list()

for (i in 1:length(the_links)) {

  the_page <- bow(the_links[i]) %>% scrape()

  temp_tibble <- tibble(

    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),

    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),

    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()

  )

  temp_list[[i]] <- temp_tibble

}

df <- as_tibble(do.call(rbind, temp_list))
```



# Static Pages — Crawling — Example

Get the source code for the next link

```
temp_list <- list()
for (i in 1:length(the_links)) {
  the_page <- bow(the_links[i]) %>% scrape()
  temp_tibble <- tibble(
    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),
    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),
    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()
  )
  temp_list[[i]] <- temp_tibble
}
df <- as_tibble(do.call(rbind, temp_list))
```

# Static Pages — Crawling — Example

Get the variables needed, put them in a tibble

```
temp_list <- list()

for (i in 1:length(the_links)) {

  the_page <- bow(the_links[i]) %>% scrape()

  temp_tibble <- tibble(

    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),

    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),

    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()

  )

  temp_list[[i]] <- temp_tibble

}
```

# Static Pages — Crawling — Example

Add each tibble into the previously-created list

```
temp_list <- list()

for (i in 1:length(the_links)) {

  the_page <- bow(the_links[i]) %>% scrape()

  temp_tibble <- tibble(

    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),

    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),

    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()

  )

  temp_list[[i]] <- temp_tibble

}

df <- as_tibble(do.call(rbind, temp_list))
```

# Static Pages — Crawling — Example

Turn the list into a tibble

```
temp_list <- list()

for (i in 1:length(the_links)) {

  the_page <- bow(the_links[i]) %>% scrape()

  temp_tibble <- tibble(

    "constituency" = the_page %>% html_elements("#constituency") %>% html_text(),

    "second_party" = the_page %>% html_element("tr:nth-child(3) td:nth-child(1)") %>%
      html_text(),

    "vote_share" = the_page %>% html_elements("tr:nth-child(3) td:nth-child(3)") %>%
      html_text()

  )

  temp_list[[i]] <- temp_tibble

}

df <- as_tibble(do.call(rbind, temp_list))
```

# Static Pages — Crawling — Example

Check the resulting dataset

```
head(df, 10)
```

```
## # A tibble: 100 x 3
##   constituency second_party vote_share
##   <chr>         <chr>         <chr>
## 1 Mühlshafen   Green          26.1%
## 2 Benwerder    Conservative 24.8%
## 3 Steffisfelden Green          25.7%
## 4 Dillon       Conservative 27%
## 5 Kilnard      Conservative 28.8%
## 6 Granderry    Labour        26.1%
## 7 Enkmelo      Liberal       26.8%
## 8 Gänsernten   Green         26.6%
## 9 Leonrau      Conservative 25%
## 10 Zotburg     Conservative 28.4%
## # ... with 90 more rows
```

# Exercise

45 : 00

10) Crawl into members' personal pages to create a rich dataset

- with members being the unit of observation

Hints:

- see an example dataset at [https://luzpar.netlify.app/files/exercises/static\\_data.csv](https://luzpar.netlify.app/files/exercises/static_data.csv)
- start with the related code in the previous slides, and adopt it to your needs
- practice with 3 members until you are ready to run the loop for all
  - e.g., by replacing `1:length(the_links)` with `1:3` for the loop

# Part 5. Scraping Dynamic Pages

[Back to the contents slide.](#)

# Dynamic Pages — Overview

- Dynamic pages are ones that display custom content
  - different visitors might see different content on the same page
    - while the URL remains the same
  - depending on, for example, their own input
    - e.g., clicks, scrolls
  - <https://luzpar.netlify.app/documents/> is a page with a dynamic part
- Dynamic pages are more difficult than static pages to scrape
  - it involves three, instead of two, steps
  - we will have a new package, R Selenium, for the additional step



# Dynamic Pages — Three Steps to Scrape

Scraping dynamic pages involves three main tasks

- Create the desired instance of the dynamic page
  - with the `RSelenium` package
  - e.g., by clicking, scrolling, filling in forms
    - from within R
- Get the source code into R
  - `RSelenium` downloads XML
  - `rvest` turns it into HTML
- Select the exact information needed from the source code
  - as for static pages
  - with the `thervest` page

# Dynamic Pages — Rseelenium — Overview

- A package that integrates **Selenium 2.0 WebDriver** into R
  - created by **John Harrison**
  - downloaded 4,874 times last month
  - last updated in February 2020
- A lot has already been written on this package
  - you will find solutions to, or help for, any issues online
  - see the **package documentation** and the **vignettes** for basic functionality
  - Google searches return code and tutorials in various languages
    - not only R but also Python, Java

# Dynamic Pages — Rseelenium — Overview

- The package involves more methods than functions
  - code look slightly unusual
  - as it follows the logic behind Selenium
- It allows interacting with two things — and it is crucial that we are aware of the difference
  - browsers
    - e.g., opening a browser and navigating to a page
  - elements
    - e.g., opening and clicking on a drop-down menu

# Interacting with Browsers

# Dynamic Pages — Browsers — Starting a Server

- Use the `rsDriver` function to start a server
  - so that you can control a web browser from within R

```
rsDriver(port = 4567L,  
        browser = "chrome",  
        version = "latest",  
        chromever = "latest",  
        ...  
)
```

# Dynamic Pages — Browsers — Starting a Server

- Use the `rsDriver` function to start a server
  - so that you can control a web browser from within R
- Note that the defaults can cause errors, such as
  - trying to start two servers from the same port

```
rsDriver(port = 4567L,  
        browser = "chrome",  
        version = "latest",  
        chromeever = "latest",  
        ...  
)
```

# Dynamic Pages — Browsers — Starting a Server

- Use the `rsDriver` function to start a server
  - so that you can control a web browser from within R
- Note that the defaults can cause errors, such as
  - trying to start two servers from the same port
  - any mismatch between the version and driver numbers

```
rsDriver(port = 4567L,  
        browser = "chrome",  
        version = "latest",  
        chromeversion = "latest",  
        ...  
)
```

# Dynamic Pages — Browsers — Starting a Server

- The latest version of the driver is too new for my browser
  - I have to use an older version to make it work
  - after checking the available versions with the following code

```
binman::list_versions("chromedriver")
```

```
## $win32  
## [1] "89.0.4389.23" "90.0.4430.24" "91.0.4472.19"
```

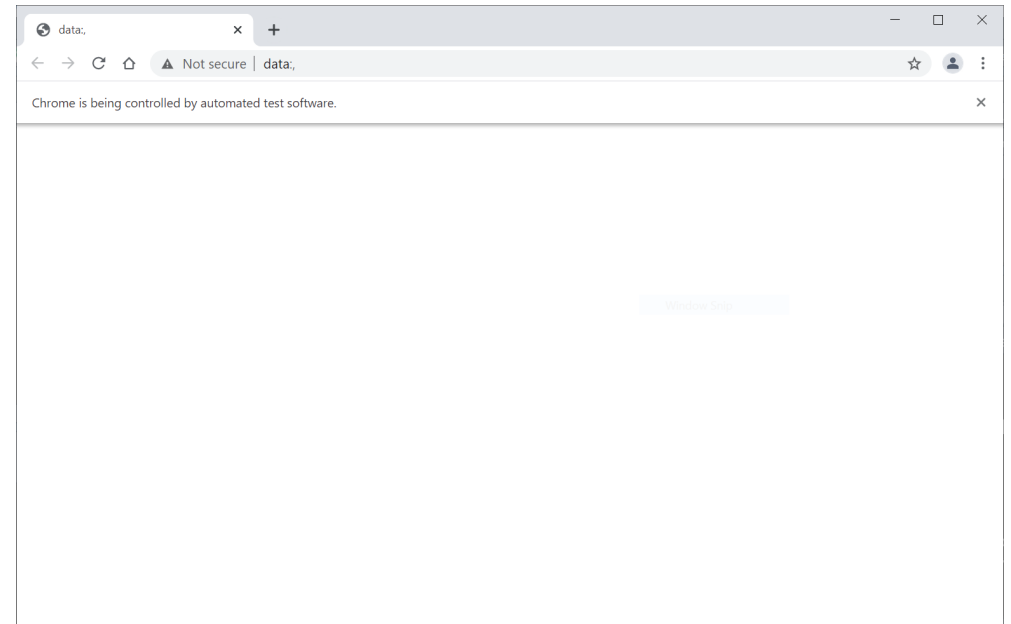
- Note that
  - you can only use the version that *you* get
  - not one of the version that you see on this slide



# Dynamic Pages — Browsers — Starting a Server

- Then the function works
  - a web browser opens as a result
  - an R object named `driver` is created
- Note that
  - the browser says "Chrome is being controlled by automated test software."
  - you should avoid controlling this browser manually
  - you should also avoid creating multiple servers

```
driver <- rsDriver(chromeversion = "90.0.4430.24")
```



# Dynamic Pages — Browsers — Starting a Server

Separate the `client` and `server` as different objects

```
browser <- driver$client  
server <- driver$server
```

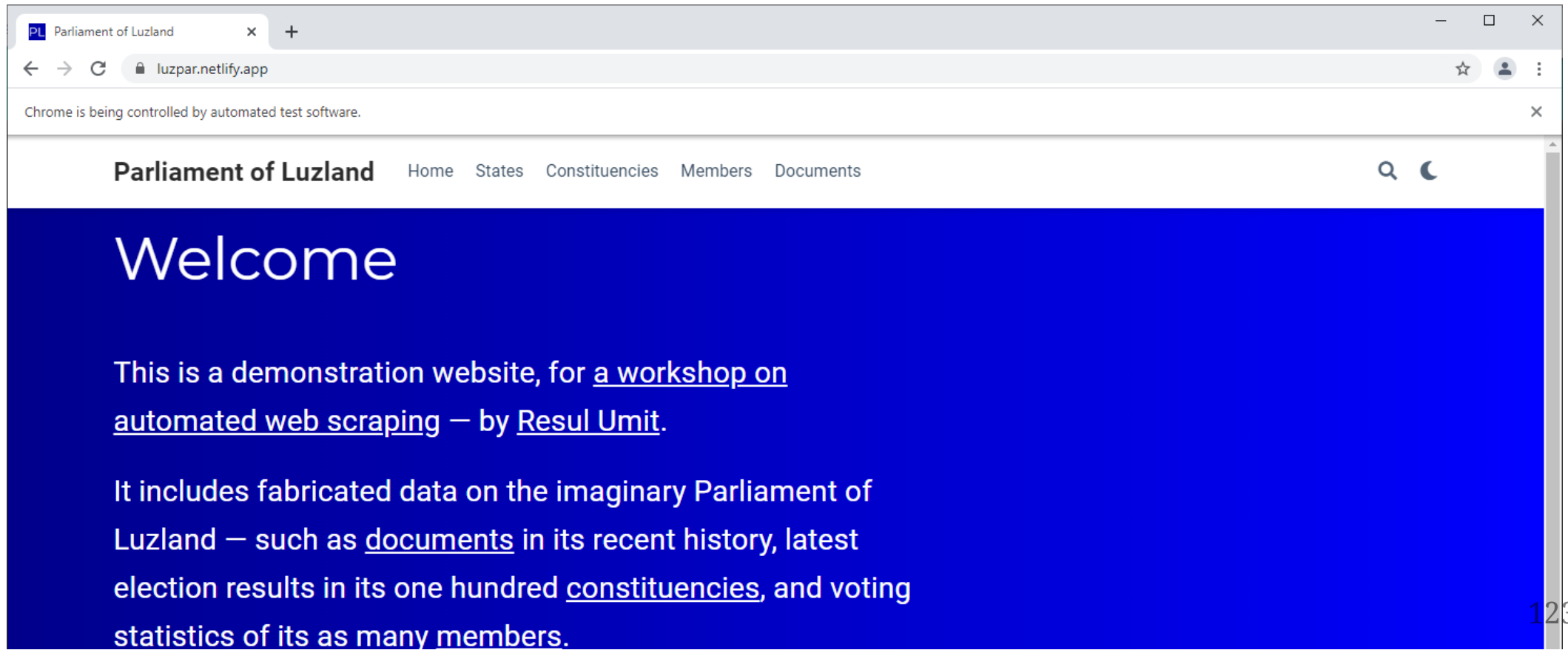
Note that

- `rsDriver()` creates a client and a server
  - the code above singles out the client, with which our code will interact
  - client is best thought as the browser itself
    - it has the class of `remoteDriver`

# Dynamic Pages — Browsers — Navigate

Navigate to a page with the following notation

```
browser$navigate(url = "https://luzpar.netlify.app")
```



# Dynamic Pages — Browsers — Navigate

Navigate to a page with the following notation

```
browser$navigate("https://luzpar.netlify.app")
```

Note that

- navigate is called **a method, not a function**
  - it cannot be piped into browser
  - use the dollar sign **\$** notation instead
  - it is not necessary to type the name of the `url` argument

# Dynamic Pages — Browsers — Navigate

Check the description of any method as follows

- with no parentheses after the method name

```
browser$navigate
```

```
Class method definition for method navigate()
function (url)
{
  "Navigate to a given url."
  qpath <- sprintf("%s/session/%s/url", serverURL, sessionInfo[["id"]])
  queryRD(qpath, "POST", qdata = list(url = url))
}
<environment: 0x00000173db9035a8>

Methods used:
  "queryRD"
```

# Dynamic Pages — Browsers — Navigate

Go back to the previous URL

```
browser$goBack()
```

Go forward

```
browser$goForward()
```

Refresh the page

```
browser$refresh()
```

# Dynamic Pages — Browsers — Navigate

Get the URL of the current page

```
browser$CurrentUrl()
```

Get the title of the current page

```
browser$getTitle()
```

# Dynamic Pages — Browsers — Close and Open

Close the browser

- which will not close the session on the server
  - recall that we singled the client out

```
browser$close()
```

Open a new browser

- which does not require the `rsDriver` function
  - because the server is still running

```
browser$open()
```



# Dynamic Pages — Browsers — Get Page Source

Get the page source

```
browser$getPageSource()[[1]]
```

# Dynamic Pages — Browsers — Get Page Source

Get the page source

```
browser$getPageSource()[[1]]
```

Note that

- this method returns a list
  - XML source is in the first item
  - this is why we need the `[[1]]` bit
- this is akin to `read_html()` for static pages
  - or `bow() %>% scrape()`
- `rvest` usually takes over after this step

# Dynamic Pages — Browsers — Get Page Source

Get the page source

- by combining the two package

```
browser$getPageSource()[[1]] %>%  
  read_html() %>%  
  html_elements("#title a") %>%  
  html_attr("href")
```

```
[1] "https://github.com/resulumit/scrp_workshop"  
[2] "https://resulumit.com/"  
[3] "https://parliament-luzland.netlify.app/documents/"  
[4] "https://parliament-luzland.netlify.app/constituencies/"  
[5] "https://parliament-luzland.netlify.app/members/"  
[6] "https://parliament-luzland.netlify.app/states/"  
[7] "https://github.com/rstudio/blogdown"  
[8] "https://gohugo.io/"  
[9] "https://github.com/wowchemy"
```

# Dynamic Pages — Browsers — Get Page Source

Get the page source

- by using both packages

```
browser$getPageSource()[[1]] %>%  
  read_html() %>%  
  html_elements("#title a") %>%  
  html_attr("href")
```

Note that

- this method gets the source for only what is physically visible on the browser
- window size and position might become important
  - you may wish to maximise the window

```
browser$browser$maxWindowSize()
```

# Dynamic Pages — Browsers — Get Page Source

Get the page source

- by using both packages

```
browser$getPageSource()[[1]] %>%  
  read_html() %>%  
  html_elements("#title a") %>%  
  html_attr("href")
```

Note that

- we still need the `read_html()` function
  - to turn XML into HTML

# Exercises

15:00

11) Navigate to and get the source code of a page

- e.g., <https://luzpar.netlify.app/constituencies/>
- by using both packages

13) See what other methods are available to interact with browsers

- by typing the object name for your client into R console
  - followed by the dollar sign
  - and hitting the tab key on your keyboard if necessary
- read the description for one or more of them

14) Try one or more new methods

- e.g., take a screenshot of your browser
  - and view it in R

# Interacting with Elements

# Dynamic Pages — Elements — Find

- Locate an element on the open browser
  - to be interacted later on
    - e.g., clicking on the element
- Note that
  - the default selector is xpath
  - requires entering the xpath value

```
findElement(using = "xpath",  
            value  
            )
```



# Dynamic Pages — Elements — Find

- Locate an element on the open browser
  - using CSS selectors
- Note that
  - typing "css", instead of "css selector", also works
  - there are other selector schemes as well, including
    - id
    - name
    - link text

```
findElement(using = "css selector",  
             value  
             )
```

# Dynamic Pages — Elements — Find — Selectors

If there were a button on a page with the following DOM...

```
<button class="big-button" id="only-button" name="clickable">Click Me</button>
```

Any of the following would find it

```
browser$findElement(using = "xpath", value = '//*[@id = "only-button"]')  
browser$findElement(using = "css selector", value = ".big-button")  
browser$findElement(using = "css", value = "#only-button")  
browser$findElement(using = "id", value = "only-button")  
browser$findElement(using = "name", value = "clickable")
```

# Dynamic Pages — Elements — Objects

Save elements as R objects to be interacted later on

```
button <- browser$findElement(using = ..., value = ...)
```

Note the difference between the classes of clients and elements

```
class(browser)
```

```
[1] "remoteDriver"  
attr(,"package")  
[1] "RSelenium"
```

```
class(button)
```

```
[1] "webElement"  
attr(,"package")  
[1] "RSelenium"
```

# Dynamic Pages — Elements — Highlight

Highlight the element found in the previous step, with the `highlightElement` method

```
# navigate to a page  
browser$navigate("http://luzpar.netlify.app/")  
  
# find the element  
menu_states <- browser$findElement(using = "link text", value = "States")  
  
# highlight it to see if we found the correct element  
menu_states$highlightElement()
```

Note that

- the highlighted element will flash for a second or two on the browser
  - helpful to check if selection worked as intended

# Dynamic Pages — Elements — Highlight

Highlight the element found in the previous step, with the `highlightElement` method

```
# navigate to a page  
browser$navigate("http://luzpar.netlify.app/")  
  
# find the element  
menu_states <- browser$findElement(using = "link text", value = "States")  
  
# highlight it to see if we found the correct element  
`menu_states$`highlightElement()
```

Note that

- the highlighted element will flash for a second or two on the browser
  - helpful to check if selection worked as intended
- the highlight method is applied to the element (`menu_states`), not to the client (`browser`)
  - compare it to the find method

# Dynamic Pages — Elements — Click

Click on the element found in the previous step, with the `clickElement` method

```
# navigate to a page  
browser$navigate("http://luzpar.netlify.app/")  
  
# find an element  
search_icon <- browser$findElement(using = "css", value = ".fa-search")  
  
# click on it  
search_icon$clickElement()
```

# Dynamic Pages — Elements — Input

- Provide input to elements, such as
  - text, with the `.yellow-h[value]` argument

```
sendKeysToElement(list(value,  
                        key  
                        )  
                  )
```

# Dynamic Pages — Elements — Input

- Provide input to elements, such as
  - text, with the `.yellow-h[value]` argument
  - keyboard presses or mouse gestures, with the `.yellow-h[key]` argument
- Note that
  - user provides values while the selenium keys are pre-defined

```
sendKeysToElement(list(value,  
                        key  
                        )  
                  )
```



# Dynamic Pages — Elements — Input — Selenium Keys

View the list of Selenium keys

```
as_tibble(selKeys) %>% names()
```

```
## [1] "null"      "cancel"    "help"      "backspace"
## [5] "tab"       "clear"     "return"    "enter"
## [9] "shift"     "control"   "alt"       "pause"
## [13] "escape"    "space"     "page_up"   "page_down"
## [17] "end"       "home"      "left_arrow" "up_arrow"
## [21] "right_arrow" "down_arrow" "insert"    "delete"
## [25] "semicolon" "equals"    "numpad_0"  "numpad_1"
## [29] "numpad_2"  "numpad_3"  "numpad_4"  "numpad_5"
## [33] "numpad_6"  "numpad_7"  "numpad_8"  "numpad_9"
## [37] "multiply"  "add"       "separator" "subtract"
## [41] "decimal"   "divide"    "f1"        "f2"
## [45] "f3"       "f4"        "f5"        "f6"
## [49] "f7"       "f8"        "f9"        "f10"
## [53] "f11"      "f12"       "command_meta"
```

# Dynamic Pages — Elements — Input — Selenium Keys — Note

Choosing the body element, you can scroll up and down a page

```
body <- browser$findElement(using = "css", value = "body")  
body$sendKeysToElement(list(key = "page_down"))
```

# Dynamic Pages — Elements — Input — Example

Search the demonstration site

```
# navigate to the home page
browser$navigate("http://luzpar.netlify.app/")

# find the search icon and click on it
search_icon <- browser$findElement(using = "css", value = ".fa-search")
search_icon$clickElement()

# find the search bar on the new page and click on it
search_bar <- browser$findElement(using = "css", value = "#search-query")
search_bar$clickElement()

# search for the keyword "Law" and click enter
search_bar$sendKeysToElement(list(value = "Law", key = "enter"))
```

# Dynamic Pages — Elements — Input — Example

Slow down the code where necessary, with the `Sys.sleep`

- for ethical reasons
- because R might be faster than the browser

```
# navigate to the home page
browser$navigate("http://luzpar.netlify.app/")

# find the search icon and click on it
search_icon <- browser$findElement(using = "css", value = ".fa-search")
search_icon$clickElement()

# sleep for 2 seconds
Sys.sleep(2)

# find the search bar on the new page and click on it
search_bar <- browser$findElement(using = "css", value = "#search-query")
search_bar$clickElement()

# search for the keyword "Law" and click enter
search_bar$sendKeysToElement(list(value = "Law", key = "enter"))
```

# Dynamic Pages — Elements — Input — Clear

Clear text, or a value, from an element

```
search_bar$clearElement()
```

# Exercise

10:00

15) Conduct an internet search programatically

- navigate to <https://duckduckgo.com/>
  - just to keep it simple as Google would require you to scroll down and accept a policy
- find, highlight, and fill in the search bar
  - hit enter

16) Scroll down programatically, and up

- to see all results

# Dynamic Pages — Elements — Switch Frames

- Switch to a different frame on a page
  - some pages have multiple frames
  - you can think of them as browsers within browsers
  - while in one frame, we cannot work with the page source of another frame

```
switchToFrame(Id  
              )
```

- Note that
  - there is one such page on the demonstration website
    - <https://luzpar.netlify.app/documents/>
    - featuring a shiny app that lives originally lives at <https://resulumit.shinyapps.io/luzpar/>
  - the Id argument takes an element object, unquoted
    - setting it to NULL returns to the default frame

# Dynamic Pages — Elements — Switch Frames

Switch to a non-default frame

```
# navigate to a page and wait for the frame to load
browser$navigate("https://luzpar.netlify.app/documents/")
Sys.sleep(4)

# find the frame, which is an element
app_frame <- browser$findElement("css", "iframe")

# switch to it
browser$switchToFrame(Id = app_frame)

#switch back to the default frame
browser$switchToFrame(Id = NULL)
```



# Dynamic Pages — Scraping — Example

## Task:

- I need to download specific documents published by the parliament
  - e.g., proposals and reports
- The related section of the website is a dynamic page
  - initially it is empty, and clicking on things do not change the URL

## Plan:

- Interact with the page until it displays the desired list of documents
- Get the page source and separate the links
- Write a for loop to
  - visit the related pages one by one
  - download the documents

# Dynamic Pages — Scraping — Example

Interact with the page until it displays the desired list of documents

```
# navigate to the desired page and wait a little
browser$navigate("https://luzpar.netlify.app/documents/")
Sys.sleep(4)

# switch to the frame with the app
app_frame <- browser$findElement("css", "iframe")
browser$switchToFrame(Id = app_frame)

# find and open the drop down menu
drop_down <- browser$findElement(using = "css", value = ".bs-placeholder")
drop_down$clickElement()

# choose proposals
proposal <- browser$findElement(using = 'css', "[id='bs-select-1-1']")
proposal$clickElement()

# choose reports
report <- browser$findElement(using = 'css', "[id='bs-select-1-2']")
report$clickElement()

# close the drop down menu
drop_down$clickElement()
```

# Dynamic Pages — Scraping — Example

Get the page source and separate the links

```
the_links <- browser$getPageSource()[[1]] %>%  
  read_html() %>%  
  html_elements("td a") %>%  
  html_attr("href")  
  
print(the_links)
```

```
## [1] "https://luzpar.netlify.app/documents/human-rights-2021/"  
## [2] "https://luzpar.netlify.app/documents/greenhouse-gas-emissions-2021/"  
## [3] "https://luzpar.netlify.app/documents/tax-reform-2020/"  
## [4] "https://luzpar.netlify.app/documents/parliamentary-staff-2020/"  
## [5] "https://luzpar.netlify.app/documents/cyber-security-2019/"  
## [6] "https://luzpar.netlify.app/documents/electronic-cigarettes-2019/"
```

# Dynamic Pages — Scraping — Example

Write a for loop to download PDFs

```
for (i in 1:length(the_links)) {  
  pdf_link <- bow(url = the_links[i]) %>%  
    scrape() %>%  
    html_elements(css = ".btn-page-header") %>%  
    html_attr("href") %>%  
    url_absolute(base = "https://luzpar.netlify.app/")  
  
  download.file(url = pdf_link, destfile = basename(pdf_link))  
}
```

# Exercise

30:00

17) Collect data on a subset of documents

- article tags and image credits
- for documents within the Law and Proposal categories
- published after 2019

Hint

- start with the related code in the previous slides, and adopt it to your needs

# References

[Back to the contents slide.](#)

# References

- Harrison, J. (2020). **RSelenium: R Bindings for 'Selenium WebDriver'**. R package, version 1.7.7.
- Meissner, P., & Ren, K. (2020). **robotstxt: A 'robots.txt' Parser and 'Webbot'/'Spider'/'Crawler' Permissions Checker**. R package, version 0.7.13.
- Perepolkin, D. (2019). **polite: Be Nice on the Web**. R package, version 0.1.1.
- Wickham, H. (2021). **rvest: Easily Harvest (Scrape) Web Pages**. R package, version 1.0.0.
- Wickham, H., François, R., Henry, L., & Müller, K. (2021). **dplyr: A grammar of data manipulation**. R package, version 1.0.6.
- Xie, Y. (2020). **xaringan: Presentation Ninja**. R package, version 0.19.

The workshop ends here.

Congratulations for making it this far, and  
thank you for joining me!

[Back to the contents slide.](#)