Package ‘rvest’

May 9, 2026

Title Easily Harvest (Scrape) Web Pages

Version 1.0.5

Description Wrappers around the 'xml2' and 'httr' packages to make it easy to download, then manipulate, HTML and XML.

License MIT + file LICENSE

URL https://rvest.tidyverse.org/, https://github.com/tidyverse/rvest

BugReports https://github.com/tidyverse/rvest/issues

Depends R (>= 4.1)

Imports cli, glue, httr (>= 0.5), lifecycle (>= 1.0.3), magrittr, rlang (>= 1.1.0), selectr, tibble, xml2 (>= 1.4.0)

Suggests chromote, covr, knitr, purrr, R6, readr, repurrrsive, rmarkdown, spelling, stringi (>= 0.3.1), testthat (>= 3.0.2), tidyr, webfakes

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

NeedsCompilation no

Author Hadley Wickham [aut, cre], Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2025-08-29 14:00:02 UTC

Contents

`html_attr` *Get element attributes*

Description

`html_attr()` gets a single attribute; `html_attrs()` gets all attributes.

Usage

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

html_attrs(x)
```

Arguments

`x`	A document (from `read_html()`), node set (from `html_elements()`), node (from `html_element()`), or session (from `session()`).
`name`	Name of attribute to retrieve.
`default`	A string used as a default value when the attribute does not exist in every element.

Value

A character vector (for `html_attr()`) or list (`html_attrs()`) the same length as `x`.

Examples

```
html <- minimal_html('<ul>
  <li><a href="https://a.com" class="important">a</a></li>
  <li class="active"><a href="https://c.com">b</a></li>
  <li><a href="https://c.com">b</a></li>
  </ul>')

html |> html_elements("a") |> html_attrs()

html |> html_elements("a") |> html_attr("href")
html |> html_elements("li") |> html_attr("class")
html |> html_elements("li") |> html_attr("class", default = "inactive")
```

---

html_children *Get element children*

Description

Get element children

Usage

```
html_children(x)
```

Arguments

x	A document (from read_html()), node set (from html_elements()), node (from html_element()), or session (from session()).

Examples

```
html <- minimal_html("<ul><li>1<li>2<li>3</ul>")
ul <- html_elements(html, "ul")
html_children(ul)

html <- minimal_html("<p>Hello <b>Hadley</b><i>!</i>")
p <- html_elements(html, "p")
html_children(p)
```

`html_element` *Select elements from an HTML document*

Description

`html_element()` and `html_elements()` find HTML element using CSS selectors or XPath expressions. CSS selectors are particularly useful in conjunction with https://selectorgadget.com/, which makes it very easy to discover the selector you need.

Usage

```
html_element(x, css, xpath)

html_elements(x, css, xpath)
```

Arguments

`x`	Either a document, a node set or a single node.
`css, xpath`	Elements to select. Supply one of `css` or `xpath` depending on whether you want to use a CSS selector or XPath 1.0 expression.

Value

`html_element()` returns a nodeset the same length as the input. `html_elements()` flattens the output so there's no direct way to map the output to the input.

CSS selector support

CSS selectors are translated to XPath selectors by the **selectr** package, which is a port of the python **cssselect** library, https://pythonhosted.org/cssselect/.

It implements the majority of CSS3 selectors, as described in https://www.w3.org/TR/2011/REC-css3-selectors-20110929/. The exceptions are listed below:

- Pseudo selectors that require interactivity are ignored: `:hover`, `:active`, `:focus`, `:target`, `:visited`.
- The following pseudo classes don't work with the wild card element, *: `*:first-of-type`, `*:last-of-type`, `*:nth-of-type`, `*:nth-last-of-type`, `*:only-of-type`
- It supports `:contains(text)`
- You can use !=, `[foo!=bar]` is the same as `:not([foo=bar])`
- `:not()` accepts a sequence of simple selectors, not just a single simple selector.

Examples

```
html <- minimal_html("
  <h1>This is a heading</h1>
  <p id='first'>This is a paragraph</p>
  <p class='important'>This is an important paragraph</p>
")

html |> html_element("h1")
html |> html_elements("p")
html |> html_elements(".important")
html |> html_elements("#first")

# html_element() vs html_elements() -------------------------------------
html <- minimal_html("
  <ul>
   <li><b>C-3PO</b> is a <i>droid</i> that weighs <span class='weight'>167 kg</span></li>
   <li><b>R2-D2</b> is a <i>droid</i> that weighs <span class='weight'>96 kg</span></li>
    <li><b>Yoda</b> weighs <span class='weight'>66 kg</span></li>
    <li><b>R4-P17</b> is a <i>droid</i></li>
  </ul>
")
li <- html |> html_elements("li")

# When applied to a node set, html_elements() returns all matching elements
# beneath any of the inputs, flattening results into a new node set.
li |> html_elements("i")

# When applied to a node set, html_element() always returns a vector the
# same length as the input, using a "missing" element where needed.
li |> html_element("i")
# and html_text() and html_attr() will return NA
li |> html_element("i") |> html_text2()
li |> html_element("span") |> html_attr("class")
```

---

html_encoding_guess *Guess faulty character encoding*

Description

`html_encoding_guess()` helps you handle web pages that declare an incorrect encoding. Use `html_encoding_guess()` to generate a list of possible encodings, then try each out by using `encoding` argument of `read_html()`. `html_encoding_guess()` replaces the deprecated `guess_encoding()`.

Usage

```
html_encoding_guess(x)
```

Arguments

x	A character vector.

Examples

```
# A file with bad encoding included in the package
path <- system.file("html-ex", "bad-encoding.html", package = "rvest")
x <- read_html(path)
x |> html_elements("p") |> html_text()

html_encoding_guess(x)
# Two valid encodings, only one of which is correct
read_html(path, encoding = "ISO-8859-1") |> html_elements("p") |> html_text()
read_html(path, encoding = "ISO-8859-2") |> html_elements("p") |> html_text()
```

`html_form` *Parse forms and set values*

Description

Use `html_form()` to extract a form, set values with `html_form_set()`, and submit it with `html_form_submit()`.

Usage

```
html_form(x, base_url = NULL)

html_form_set(form, ...)

html_form_submit(form, submit = NULL)
```

Arguments

`x`	A document (from `read_html()`), node set (from `html_elements()`), node (from `html_element()`), or session (from `session()`).
`base_url`	Base url of underlying HTML document. The default, `NULL`, uses the url of the HTML document underlying `x`.
`form`	A form
`...`	<`dynamic-dots`> Name-value pairs giving fields to modify. Provide a character vector to set multiple checkboxes in a set or select multiple values from a multi-select.
`submit`	Which button should be used to submit the form? • `NULL`, the default, uses the first button. • A string selects a button by its name. • A number selects a button using its relative position.

Value

- `html_form()` returns as S3 object with class `rvest_form` when applied to a single element. It returns a list of `rvest_form` objects when applied to multiple elements or a document.
- `html_form_set()` returns an `rvest_form` object.
- `html_form_submit()` submits the form, returning an httr response which can be parsed with `read_html()`.

See Also

HTML 4.01 form specification: https://www.w3.org/TR/html401/interact/forms.html

Examples

```
html <- read_html("http://www.google.com")
search <- html_form(html)[[1]]

search <- search |> html_form_set(q = "My little pony", hl = "fr")

# Or if you have a list of values, use !!!
vals <- list(q = "web scraping", hl = "en")
search <- search |> html_form_set(!!!vals)

# To submit and get result:
## Not run:
resp <- html_form_submit(search)
read_html(resp)

## End(Not run)
```

---

html_name *Get element name*

Description

Get element name

Usage

```
html_name(x)
```

Arguments

x	A document (from read_html()), node set (from html_elements()), node (from html_element()), or session (from session()).

Value

A character vector the same length as x

Examples

```
url <- "https://rvest.tidyverse.org/articles/starwars.html"
html <- read_html(url)

html |>
  html_element("div") |>
  html_children() |>
  html_name()
```

`html_table` *Parse an html table into a data frame*

Description

The algorithm mimics what a browser does, but repeats the values of merged cells in every cell that cover.

Usage

```
html_table(
  x,
  header = NA,
  trim = TRUE,
  fill = deprecated(),
  dec = ".",
  na.strings = "NA",
  convert = TRUE
)
```

Arguments

`x`	A document (from `read_html()`), node set (from `html_elements()`), node (from `html_element()`), or session (from `session()`).
`header`	Use first row as header? If `NA`, will use first row if it consists of `<th>` tags. If `TRUE`, column names are left exactly as they are in the source document, which may require post-processing to generate a valid data frame.
`trim`	Remove leading and trailing whitespace within each cell?
`fill`	Deprecated - missing cells in tables are now always automatically filled with `NA`.
`dec`	The character used as decimal place marker.
`na.strings`	Character vector of values that will be converted to `NA` if `convert` is `TRUE`.
`convert`	If `TRUE`, will run `type.convert()` to interpret texts as integer, double, or `NA`.

Value

When applied to a single element, `html_table()` returns a single tibble. When applied to multiple elements or a document, `html_table()` returns a list of tibbles.

Examples

```
sample1 <- minimal_html("<table>
  <tr><th>Col A</th><th>Col B</th></tr>
  <tr><td>1</td><td>x</td></tr>
  <tr><td>4</td><td>y</td></tr>
  <tr><td>10</td><td>z</td></tr>
</table>")
```

```
sample1 |>
  html_element("table") |>
  html_table()

# Values in merged cells will be duplicated
sample2 <- minimal_html("<table>
  <tr><th>A</th><th>B</th><th>C</th></tr>
  <tr><td>1</td><td>2</td><td>3</td></tr>
  <tr><td colspan='2'>4</td><td>5</td></tr>
  <tr><td>6</td><td colspan='2'>7</td></tr>
</table>")
sample2 |>
  html_element("table") |>
  html_table()

# If a row is missing cells, they'll be filled with NAs
sample3 <- minimal_html("<table>
  <tr><th>A</th><th>B</th><th>C</th></tr>
  <tr><td colspan='2'>1</td><td>2</td></tr>
  <tr><td colspan='2'>3</td></tr>
  <tr><td>4</td></tr>
</table>")
sample3 |>
  html_element("table") |>
  html_table()
```

`html_text` *Get element text*

Description

There are two ways to retrieve text from a element: `html_text()` and `html_text2()`. `html_text()` is a thin wrapper around `xml2::xml_text()` which returns just the raw underlying text. `html_text2()` simulates how text looks in a browser, using an approach inspired by JavaScript's innerText(). Roughly speaking, it converts `<br />` to `"\n"`, adds blank lines around `<p>` tags, and lightly formats tabular data.

`html_text2()` is usually what you want, but it is much slower than `html_text()` so for simple applications where performance is important you may want to use `html_text()` instead.

Usage

```
html_text(x, trim = FALSE)

html_text2(x, preserve_nbsp = FALSE)
```

Arguments

`x`	A document, node, or node set.
`trim`	If `TRUE` will trim leading and trailing spaces.

preserve_nbsp Should non-breaking spaces be preserved? By default, `html_text2()` converts to ordinary spaces to ease further computation. When `preserve_nbsp` is `TRUE`, ` ` will appear in strings as `"\ua0"`. This often causes confusion because it prints the same way as `" "`.

Value

A character vector the same length as `x`

Examples

```
# To understand the difference between html_text() and html_text2()
# take the following html:

html <- minimal_html(
  "<p>This is a paragraph.
    This another sentence.<br>This should start on a new line"
)

# html_text() returns the raw underlying text, which includes whitespace
# that would be ignored by a browser, and ignores the <br>
html |> html_element("p") |> html_text() |> writeLines()

# html_text2() simulates what a browser would display. Non-significant
# whitespace is collapsed, and <br> is turned into a line break
html |> html_element("p") |> html_text2() |> writeLines()

# By default, html_text2() also converts non-breaking spaces to regular
# spaces:
html <- minimal_html("<p>x y</p>")
x1 <- html |> html_element("p") |> html_text()
x2 <- html |> html_element("p") |> html_text2()

# When printed, non-breaking spaces look exactly like regular spaces
x1
x2
# But aren't actually the same:
x1 == x2
# Which you can confirm by looking at their underlying binary
# representaion:
charToRaw(x1)
charToRaw(x2)
```

LiveHTML *Interact with a live web page*

Description

[Experimental]

You construct an LiveHTML object with `read_html_live()` and then interact, like you're a human, using the methods described below. When debugging a scraping script it is particularly useful to use `$view()`, which will open a live preview of the site, and you can actually see each of the operations performed on the real site.

rvest provides relatively simple methods for scrolling, typing, and clicking. For richer interaction, you probably want to use a package that exposes a more powerful user interface, like selendir.

Public fields

`session` Underlying chromote session object. For expert use only.

Methods

Public methods:

- `LiveHTML$new()`
- `LiveHTML$print()`
- `LiveHTML$view()`
- `LiveHTML$html_elements()`
- `LiveHTML$click()`
- `LiveHTML$get_scroll_position()`
- `LiveHTML$scroll_into_view()`
- `LiveHTML$scroll_to()`
- `LiveHTML$scroll_by()`
- `LiveHTML$type()`
- `LiveHTML$press()`
- `LiveHTML$clone()`

Method `new()`**:** initialize the object

Usage:

```
LiveHTML$new(url)
```

Arguments:

`url` URL to page.

Method `print()`**:** Called when `print()`ed

Usage:

```
LiveHTML$print(...)
```

Arguments:

`...` Ignored

Method `view()`**:** Display a live view of the site

Usage:

```
LiveHTML$view()
```

Method `html_elements()`**:** Extract HTML elements from the current page.

Usage:

```
LiveHTML$html_elements(css, xpath)
```

Arguments:

`css, xpath` CSS selector or xpath expression.

Method `click()`**:** Simulate a click on an HTML element.

Usage:

```
LiveHTML$click(css, n_clicks = 1)
```

Arguments:

`css` CSS selector.

`n_clicks` Number of clicks

Method `get_scroll_position()`**:** Get the current scroll position.

Usage:

```
LiveHTML$get_scroll_position()
```

Method `scroll_into_view()`**:** Scroll selected element into view.

Usage:

```
LiveHTML$scroll_into_view(css)
```

Arguments:

`css` CSS selector.

Method `scroll_to()`**:** Scroll to specified location

Usage:

```
LiveHTML$scroll_to(top = 0, left = 0)
```

Arguments:

`top, left` Number of pixels from top/left respectively.

Method `scroll_by()`**:** Scroll by the specified amount

Usage:

```
LiveHTML$scroll_by(top = 0, left = 0)
```

Arguments:

`top, left` Number of pixels to scroll up/down and left/right respectively.

Method `type()`**:** Type text in the selected element

Usage:

```
LiveHTML$type(css, text)
```

Arguments:

`css` CSS selector.

`text` A single string containing the text to type.

Method `press()`**:** Simulate pressing a single key (including special keys).

Usage:

```
LiveHTML$press(css, key_code, modifiers = character())
```

Arguments:

`css` CSS selector.

`key_code` Name of key. You can see a complete list of known keys at https://pptr.dev/api/puppeteer.keyinput.

`modifiers` A character vector of modifiers. Must be one or more of `"Shift"`, `"Control"`, `"Alt"`, or `"Meta"`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LiveHTML$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
# To retrieve data for this paginated site, we need to repeatedly push
# the "Load More" button
sess <- read_html_live("https://www.bodybuilding.com/exercises/finder")
sess$view()

sess |> html_elements(".ExResult-row") |> length()
sess$click(".ExLoadMore-btn")
sess |> html_elements(".ExResult-row") |> length()
sess$click(".ExLoadMore-btn")
sess |> html_elements(".ExResult-row") |> length()

## End(Not run)
```

`read_html` *Static web scraping (with xml2)*

Description

`read_html()` works by performing a HTTP request then parsing the HTML received using the xml2 package. This is "static" scraping because it operates only on the raw HTML file. While this works for most sites, in some cases you will need to use `read_html_live()` if the parts of the page you want to scrape are dynamically generated with javascript.

Generally, we recommend using `read_html()` if it works, as it will be faster and more robust, as it has fewer external dependencies (i.e. it doesn't rely on the Chrome web browser installed on your computer.)

Usage

```
read_html(
  x,
  encoding = "",
  ...,
  options = c("RECOVER", "NOERROR", "NOBLANKS", "HUGE")
)
```

Arguments

x	Usually a string representing a URL. See `xml2::read_html()` for other options.
encoding	Specify a default encoding for the document. Unless otherwise specified XML documents are assumed to be in UTF-8 or UTF-16. If the document is not UTF-8/16, and lacks an explicit encoding directive, this allows you to supply a default.
...	Additional arguments passed on to methods.
options	Set parsing options for the libxml2 parser. Zero or more of

RECOVER recover on errors

NOENT substitute entities

DTDLOAD load the external subset

DTDATTR default DTD attributes

DTDVALID validate with the DTD

NOERROR suppress error reports

NOWARNING suppress warning reports

PEDANTIC pedantic error reporting

NOBLANKS remove blank nodes

SAX1 use the SAX1 interface internally

XINCLUDE Implement XInclude substitution

NONET Forbid network access

NODICT Do not reuse the context dictionary

NSCLEAN remove redundant namespaces declarations

NOCDATA merge CDATA as text nodes

NOXINCNODE do not generate XINCLUDE START/END nodes

COMPACT compact small text nodes; no modification of the tree allowed afterwards (will possibly crash if you try to modify the tree)

OLD10 parse using XML-1.0 before update 5

NOBASEFIX do not fixup XINCLUDE xml:base uris

HUGE relax any hardcoded limit from the parser

OLDSAX parse using SAX2 interface before 2.7.0

IGNORE_ENC ignore internal document encoding hint

BIG_LINES Store big lines numbers in text PSVI field

Examples

```
# Start by reading a HTML page with read_html():
starwars <- read_html("https://rvest.tidyverse.org/articles/starwars.html")

# Then find elements that match a css selector or XPath expression
# using html_elements(). In this example, each <section> corresponds
# to a different film
films <- starwars |> html_elements("section")
films

# Then use html_element() to extract one element per film. Here
# we the title is given by the text inside <h2>
title <- films |>
  html_element("h2") |>
  html_text2()
title

# Or use html_attr() to get data out of attributes. html_attr() always
# returns a string so we convert it to an integer using a readr function
episode <- films |>
  html_element("h2") |>
  html_attr("data-id") |>
  readr::parse_integer()
episode
```

`read_html_live` *Live web scraping (with chromote)*

Description

[Experimental]

`read_html()` operates on the HTML source code downloaded from the server. This works for most websites but can fail if the site uses javascript to generate the HTML. `read_html_live()` provides an alternative interface that runs a live web browser (Chrome) in the background. This allows you to access elements of the HTML page that are generated dynamically by javascript and to interact with the live page by clicking on buttons or typing in forms.

Behind the scenes, this function uses the chromote package, which requires that you have a copy of Google Chrome installed on your machine.

Usage

```
read_html_live(url)
```

Arguments

`url`	Website url to read from.

Value

`read_html_live()` returns an R6 LiveHTML object. You can interact with this object using the usual rvest functions, or call its methods, like `$click()`, `$scroll_to()`, and `$type()` to interact with the live page like a human would.

Examples

```
## Not run:
# When we retrieve the raw HTML for this site, it doesn't contain the
# data we're interested in:
static <- read_html("https://www.forbes.com/top-colleges/")
static |> html_element("table")

# Instead, we need to run the site in a real web browser, causing it to
# download a JSON file and then dynamically generate the html:
dynamic <- read_html_live("https://www.forbes.com/top-colleges/")
# You may need to click the cookie consent banner if it appears
dynamic$view()

# Now we can find the table
dynamic |> html_element("table")

# And extract data from it
dynamic |>
  html_element("table") |>
  html_table()

## End(Not run)
```

---

session *Simulate a session in web browser*

Description

This set of functions allows you to simulate a user interacting with a website, using forms and navigating from page to page.

- Create a session with `session(url)`
- Navigate to a specified url with `session_jump_to()`, or follow a link on the page with `session_follow_link()`.
- Submit an html_form with `session_submit()`.
- View the history with `session_history()` and navigate back and forward with `session_back()` and `session_forward()`.
- Extract page contents with `html_element()` and `html_elements()`, or get the complete HTML document with `read_html()`.
- Inspect the HTTP response with `httr::cookies()`, `httr::headers()`, and `httr::status_code()`.

Usage

```
session(url, ...)

is.session(x)

session_jump_to(x, url, ...)

session_follow_link(x, i, css, xpath, ...)

session_back(x)

session_forward(x)

session_history(x)

session_submit(x, form, submit = NULL, ...)
```

Arguments

`url`	A URL, either relative or absolute, to navigate to.
`...`	Any additional httr config to use throughout the session.
`x`	A session.
`i`	A integer to select the ith link or a string to match the first link containing that text (case sensitive).
`css`, `xpath`	Elements to select. Supply one of `css` or `xpath` depending on whether you want to use a CSS selector or XPath 1.0 expression.
`form`	An html_form to submit
`submit`	Which button should be used to submit the form? • `NULL`, the default, uses the first button. • A string selects a button by its name. • A number selects a button using its relative position.

Examples

```
s <- session("http://hadley.nz")
s |>
  session_jump_to("hadley.jpg") |>
  session_jump_to("/") |>
  session_history()

s |>
  session_jump_to("hadley.jpg") |>
  session_back() |>
  session_history()


s |>
```

```
session_follow_link(css = "p a") |>
html_elements("p")
```

Index