Package ‘zuhtml’

November 19, 2026

Title Parse HTML with a Bundled 'Gumbo' Parser

Version 0.1.1

Description Parses real-world HTML with a bundled copy of the 'Gumbo' parser (<https://codeberg.org/gumbo-parser/gumbo-parser>), which follows the parsing algorithm of the HTML standard, so that no system library is required. Documents become immutable trees navigated with a documented subset of CSS selectors. Attributes, text, lists, tables, links, forms and page metadata (JSON-LD, microdata) are extracted into ordinary character vectors, lists and data frames, and nodes convert to Markdown. Input is a string, raw bytes, a file, a URL or a connection, and raw input is decoded as browsers decode it, from a byte-order mark or a meta element declaring the character set. Parsing is bounded by limits on input size, native memory and nesting depth.

License MIT + file LICENSE

Copyright file inst/COPYRIGHTS

URL https://github.com/pedrobtz/zuhtml, https://pedrobtz.github.io/zuhtml/

BugReports https://github.com/pedrobtz/zuhtml/issues

Depends R (>= 4.1)

Suggests commonmark, jsonlite, knitr, rmarkdown, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-GB

Config/roxygen2/version 8.1.1

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph], Google Inc. [cph] (Gumbo, bundled in src/vendor/gumbo), Bjoern Hoehrmann [cph] (UTF-8 decoder in src/vendor/gumbo/utf8.c)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Repository CRAN

Date/Publication 2026-11-19 11:20:02 UTC

Contents

html_attr 2
html_children 4
html_closest 5
html_elements 6
html_forms 8
html_fragment 9
html_info 10
html_json_ld 11
html_limits 12
html_links 14
html_list 14
html_markdown 15
html_meta 17
html_microdata 18
html_name 19
html_parse 20
html_problems 22
html_serialize 23
html_strings 24
html_table 25
html_table_cells 27
html_text 29
html_text_clean 29
html_title 31
html_url 31
zuhtml-conditions 32
zuhtml_info 33

Index **35**

`html_attr` *Attributes of elements*

Description

`html_attr()` reads one attribute from every node; `html_attrs()` reads all of them; `html_classes()` splits the `class` attribute into tokens. Values are decoded (`&` becomes `&`). An attribute that is present but empty is `""`, distinct from an absent one: test a boolean attribute such as `disabled` by presence, `!is.na(html_attr(x, "disabled"))`.

Usage

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

html_attrs(x)

html_classes(x)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
name	The attribute name: a single string.
default	The value for nodes that lack the attribute, including nodes that are not elements: a single string, possibly NA.

Details

Attribute names on HTML elements match regardless of ASCII case, as in a browser; on SVG and MathML elements they match exactly (viewBox). Namespaced attributes in foreign content are named with their prefix, as in "xlink:href". Where an element has the same attribute more than once, the first wins, as the HTML parser decides.

Value

- html_attr(): a character vector as long as x; NA for missing nodes.
- html_attrs(): a list as long as x of named character vectors, in source order; NA_character_ for missing nodes.
- html_classes(): a list as long as x of character vectors of class tokens; NA_character_ for missing nodes.

See Also

Other node values: html_markdown(), html_name(), html_serialize(), html_strings(), html_text(), html_text_clean()

Examples

```
doc <- html_parse(
  "<a href='/x' class='btn  primary' data-id=7>Go</a><input disabled>"
)
body <- html_children(html_root(doc))[2]
nodes <- html_children(body)
html_attr(nodes, "href")
html_attr(nodes, "href", default = "")
html_attrs(nodes)
html_classes(nodes)
!is.na(html_attr(nodes, "disabled"))
```

`html_children` *Navigate a document tree*

Description

Move from nodes to related nodes. Operations that return exactly one node per input node – `html_parent()`, `html_next_sibling()` and `html_previous_sibling()` – are *aligned*: the result has the same length as `x`, with a missing node where there is no answer, so that results line up with their inputs. Operations that can return many nodes per input – `html_children()`, `html_ancestors()` and `html_template_content()` – return the set of all results, without duplicates, in document order.

Usage

```
html_children(x, elements_only = TRUE)

html_parent(x)

html_ancestors(x)

html_next_sibling(x, elements_only = TRUE)

html_previous_sibling(x, elements_only = TRUE)

html_template_content(x)

html_root(x)

html_document(x)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`. A document stands for its document node.
`elements_only`	If `TRUE`, consider element nodes only, skipping text, comments and other nodes.

Details

A `<template>` element's contents are its children here, but they are inert: searches and text extraction do not descend into them. `html_template_content()` returns them explicitly.

Value

- `html_children()`, `html_ancestors()`, `html_template_content()`: a `zuhtml_nodeset` in document order. `html_ancestors()` does not include the document node.
- `html_parent()`, `html_next_sibling()`, `html_previous_sibling()`: a `zuhtml_nodeset` as long as `x`, with missing nodes where there is no parent or sibling.

- `html_root()`: a `zuhtml_nodeset` of length one: the `<html>` element of a document, or the fragment node of a fragment.
- `html_document()`: the `zuhtml_document` that owns `x`.

Nodesets

A `zuhtml_nodeset` holds node IDs that only mean something in the document that owns them. Subsetting (`[`, `[[`), `rev()`, `rep()`, `unique()` and `length<-` keep the owner; `length<-` pads with missing nodes. `c()`, `[<-` and `[[<-` accept only nodes of the same document (and, for assignment, `NA` for a missing node), and raise a `zuhtml_input_error` otherwise.

`==` and `!=` between two nodesets of the same document compare node identity, element by element: two `<p>a</p>` elements are different nodes. Comparing nodes of different documents, or a nodeset with anything else, is a `zuhtml_input_error`. `match()`, `%in%`, `setdiff()` and the other set operations compare nodes too, so a node is never found among another document's nodes. `sort()`, `order()` and `xtfrm()` follow document order.

Every other operator (arithmetic, `<` and the other orderings, `!`, `&`), the math functions, `sum()`, `max()`, `range()` and the rest of the Summary group, `mean()`, `median()` and `diff()` are refused with a `zuhtml_input_error`, and so is `rank()`, which needs `>`. Use `as.integer()` to work with the IDs as numbers.

See Also

Other navigation: `html_closest()`, `html_elements()`

Examples

```
doc <- html_parse("<ul><li>One</li><li>Two <b>bold</b></li></ul>")
body <- html_children(html_root(doc))[2]
items <- html_children(html_children(body))
items
html_parent(items)
html_next_sibling(items)
html_ancestors(html_children(items[2]))

# Template contents are reached explicitly:
tmpl <- html_parse("<template><p>inert</p></template>")
html_template_content(html_children(html_children(html_root(tmpl))[1]))
```

---

html_closest *Nearest ancestor matching a selector*

Description

For each node, the node itself if it matches `css`, otherwise its nearest ancestor element that does, as the DOM's `closest()` and jQuery's `.closest()`. Use it to go from a cell or a link to the row, card or section that contains it. `:scope` is the node itself.

Usage

```
html_closest(x, css)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
css	A CSS selector: a single string, of at most the max_selector_length bytes of the html_limits() the document was parsed with.

Value

A zuhtml_nodeset as long as x: for each node, the matching element, or a missing node where there is none (and for missing nodes).

See Also

Other navigation: html_children(), html_elements()

Examples

```
doc <- html_parse(paste0(
  "<table><tr id=a><td>1<td><b>x</b></tr>",
  "<tr id=b><td>2<td><b>y</b></tr></table>"
))
bold <- html_elements(doc, "b")
html_attr(html_closest(bold, "tr"), "id")
html_closest(bold, "ul")
```

---

html_elements *Select elements with CSS selectors*

Description

html_elements() finds every element matching a selector below the given nodes; html_element() finds the first below each node, keeping one result per input so that extracted columns stay aligned. html_matches() tests nodes themselves, and html_filter() keeps the ones that match.

Usage

```
html_elements(x, css)

html_element(x, css)

html_matches(x, css)

html_filter(x, css)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
css	A CSS selector: a single string, of at most the max_selector_length bytes of the html_limits() the document was parsed with.

Details

Searching a document includes its <html> element. Searching below an element excludes the element itself unless the selector uses :scope for it, as in ":scope > li" or ":scope.active". Matching elsewhere in the selector is against the whole document, as in a browser: "div p" below a <section> finds a <p> whose ancestor is outside the section. Template contents are never searched.

Value

- html_elements(): a zuhtml_nodeset of the matching elements below any node of x, without duplicates, in document order.
- html_element(): a zuhtml_nodeset as long as x: for each node, the first matching element below it in document order, or a missing node.
- html_matches(): a logical vector as long as x; NA for missing nodes, FALSE for nodes that are not elements.
- html_filter(): the nodes of x that match, in their order.

Supported selectors

A deliberately small subset of CSS Selectors Level 4, and nothing else:

- type (p) and universal (*) selectors;
- #id and .class;
- attributes: [a], [a=v], [a~=v], [a|=v], [a^=v], [a$=v] and [a*=v], with an optional i or s flag, as in [type=text i];
- combinators: descendant (space), child (>), next sibling (+) and later sibling (~), and selector lists (,);
- :scope, :root, :empty, :first-child, :last-child, :only-child, :nth-child(an+b), :nth-of-type(an+b), and :not() with one compound selector.

Anything else – :has(), other pseudo-classes, pseudo-elements, namespace prefixes, the of S form of :nth-child() – is an error of class zuhtml_selector_error, never a silent partial match.

Case follows HTML documents in a browser: element and attribute names match HTML elements regardless of ASCII case and SVG or MathML elements exactly; IDs and classes are case-sensitive; attribute values are case-sensitive except for the attributes HTML lists as case-insensitive (such as type and lang), and an i or s flag overrides both. Structural pseudo-classes count element siblings only. :empty is true for an element with no element or text children; comments do not count, whitespace does.

See Also

Other navigation: html_children(), html_closest()

Examples

```
doc <- html_parse(paste0(
  "<div class=card><h2>Tea</h2><span class=price>3.50</span></div>",
  "<div class=card><h2>Cake</h2></div>"
))
cards <- html_elements(doc, ".card")
html_text(html_element(cards, "h2"))
html_text(html_element(cards, ".price"))
html_elements(doc, "div > h2:first-child")
html_matches(html_elements(doc, "div, h2"), ".card")
html_filter(html_elements(doc, "div, h2"), "h2")

try(html_elements(doc, "div:has(h2)"))
```

`html_forms` *Forms and their controls*

Description

Describes each `<form>` below the given nodes (and the nodes themselves) and the controls it owns, for inspection. Nothing is submitted, fetched or built into a request.

Usage

```
html_forms(x)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.

Details

A control belongs to a form as the HTML standard's form owner says: the form named by its `form` attribute (the element with that ID, if it is a `<form>`; no form otherwise), else its nearest ancestor `<form>`. Old pages often write `<table><form><tr><td><input ...>`, where the parser closes the form at once but browsers still associate the following controls with it; such a form, left empty inside a table, owns the controls without another owner that follow it in that table, up to the next form.

Controls are `<input>`, `<select>`, `<textarea>` and `<button>` elements, in document order. Only HTML elements count: an element of those names inside `<svg>` or `<math>` is not a form or a control. Values follow the DOM:

- `type` is an input's `type` (lowercased; `"text"` when missing or unknown), `"select-one"` or `"select-multiple"`, `"textarea"`, or a button's `type` (`"submit"` by default);
- `value` is the `value` attribute (`"on"` for a checkbox or radio button without one, else `""`); a textarea's text; or a select's first selected option's value, where a single select with no option marked selected selects its first enabled option if it shows one row (its `size` is absent, invalid or 1), and none otherwise;

- `disabled` is `TRUE` for a control with a `disabled` attribute or inside a disabled `<fieldset>` (except in its first `<legend>`).

Value

A list with one element per form, in document order, each a list of:

- `action`: the `action` attribute resolved as `html_url()` resolves URLs; without one, the document's base URL, as browsers submit to the page itself; `NA` when neither resolves;
- `method`: `"get"`, `"post"` or `"dialog"` (`"get"` when missing or invalid);
- `enctype`: `"application/x-www-form-urlencoded"` (the default), `"multipart/form-data"` or `"text/plain"`;
- `id`, `name`: the attributes, or `NA`;
- `fields`: a data frame with one row per control and columns `name` (`NA` when absent; repeated names are kept), `type`, `value`, `checked` (`TRUE`/`FALSE` for checkboxes and radio buttons, `NA` otherwise), `disabled`, and `options`, a list-column holding for each select a data frame of its options (`value`, `label`, `selected`, `disabled`) and `NULL` for other controls.

See Also

Other extraction: `html_links()`, `html_list()`, `html_table()`, `html_table_cells()`, `html_url()`

Examples

```
doc <- html_parse(paste0(
  "<form action='/search' method=POST>",
  "<input name=q value='zuhtml'>",
  "<select name=sort><option>relevance<option selected>date</select>",
  "<label><input type=checkbox name=exact checked> Exact</label>",
  "<button>Search</button></form>",
  "<input form=search-options name=page value=2>",
  "<form id=search-options></form>"
), base_url = "https://example.org/")
forms <- html_forms(doc)
forms[[1]][c("action", "method")]
forms[[1]]$fields[, c("name", "type", "value", "checked")]
forms[[1]]$fields$options[[2]]
forms[[2]]$fields$name
```

`html_fragment` *Parse an HTML fragment*

Description

Parses markup as the contents of a given element, as a browser does for `innerHTML`. The context matters: `"<tr><td>x"` is a table row in a `"tbody"` context but only text in a `"div"`.

Usage

```
html_fragment(x, context = "div", ...)
```

Arguments

x	One string, or a raw vector of encoded bytes.
context	The name of the context element, an HTML element the bundled parser knows, such as "div", "tbody", "select" or "template".
...	Arguments passed on to html_parse(): encoding, base_url, comments and limits.

Value

A zuhtml_document whose document node is a fragment: html_type() reports "fragment", and html_root() returns the fragment node, whose children are the parsed nodes.

See Also

html_parse() for whole documents.

Other parsing: html_info(), html_limits(), html_parse(), html_problems(), zuhtml_info()

Examples

```
frag <- html_fragment("<tr><td>1<td>2", context = "tbody")
html_children(html_root(frag))
html_fragment("<tr><td>1<td>2")
```

---

html_info *Information about a parsed document*

Description

Information about a parsed document

Usage

```
html_info(x)
```

Arguments

x	A zuhtml_document, or a zuhtml_nodeset for the document that owns it.

Value

An object of class `zuhtml_doc_info`: a list with

- `type`: `"document"` or `"fragment"`;
- `context`: a fragment's context element, `NA` for a document;
- `nodes`, `attributes`: counts;
- `native_bytes`: memory the document owns, outside R's heap;
- `parse_peak_bytes`: the most native memory the parse held at once, counted as `max_memory` counts it, so it is the smallest `max_memory` that parses the same input with the same options;
- `input_bytes`: size of the decoded UTF-8 input;
- `encoding`: the encoding the input was decoded from;
- `encoding_source`: where that encoding came from: `"bom"` (a byte-order mark), `"argument"` (the `encoding` argument), `"meta"` (a `<meta>` declaration), `"default"` (none of these, so UTF-8), or `"string"` for character input, which is already decoded;
- `base_url`: the `base_url` given to `html_parse()`, or `NA`;
- `quirks_mode`: `"no-quirks"`, `"quirks"` or `"limited-quirks"`, as the doctype selected;
- `problems`: the number of parse problems kept, and `problems_truncated`: whether more occurred than `max_errors`.

See Also

Other parsing: `html_fragment()`, `html_limits()`, `html_parse()`, `html_problems()`, `zuhtml_info()`

Examples

```
html_info(html_parse("<!DOCTYPE html><title>t</title><p>Hello"))
```

---

html_json_ld *JSON-LD blocks*

Description

The structured data pages embed as JSON-LD: the text of each `<script type="application/ld+json">` below the given nodes (and the nodes themselves), in document order; an SVG or MathML `<script>` does not count. The type is matched as a MIME type: case-insensitively, ignoring surrounding whitespace and any parameters.

Usage

```
html_json_ld(x, parse = FALSE)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
parse	If TRUE, parse each block with jsonlite::parse_json() (with simplifyVector = FALSE, so objects are named lists and arrays are lists). The text is only ever parsed as JSON, never read as a file name or URL. This needs the jsonlite package. A wrapper around the whole block, <![CDATA[...]]> or <!-- ... -->, optionally commented out with // or /* */ as pages often do, is removed first.

Value

With parse = FALSE, a character vector: the text of each block, exactly as written. With parse = TRUE, a list as long as that vector: each block's parsed value, or NULL for a block that is not valid JSON; the texts are kept in its attribute "json".

See Also

Other metadata: html_meta(), html_microdata(), html_title()

Examples

```
doc <- html_parse(paste0(
  "<script type='application/ld+json'>",
  '{"@context": "https://schema.org", "@type": "Person", "name": "Ada"}',
  "</script>"
))
html_json_ld(doc)
if (requireNamespace("jsonlite", quietly = TRUE)) {
  html_json_ld(doc, parse = TRUE)[[1]]$name
}
```

---

html_limits *Resource limits for parsing and extraction*

Description

Every parse and extraction runs under explicit limits, so that untrusted HTML cannot exhaust memory or time. Limits are per call, not global options: pass the result to the limits argument of html_parse() and the extraction functions. Exceeding one raises a zuhtml_limit_error (see zuhtml-conditions) after all native memory is released.

Usage

```
html_limits(
  max_input = 16 * 1024^2,
  max_memory = 512 * 1024^2,
  max_depth = 512,
  max_nodes = 4e+06,
```

```
  max_errors = 100,
  max_table_cells = 1e+06,
  max_selector_length = 16 * 1024
)
```

Arguments

max_input	Largest input, in bytes of UTF-8 after decoding. Raw input is also refused, before decoding, when it is longer than four times this.
max_memory	Largest native memory the parser may hold at once, in bytes. A parse that runs past it reports the memory in use where it stopped, which the parse needs at least; once a parse completes, `html_info()$parse_peak_bytes` is the smallest value that parses that input.
max_depth	Deepest nesting of open elements. Parsing time grows with the square of nesting depth, so this bounds it; it is enforced while parsing, not afterwards.
max_nodes	Most nodes a document may have.
max_errors	Most parse problems kept for `html_problems()`. Parsing continues past it; only the record is truncated.
max_table_cells	Most cells a table may expand to, spans included.
max_selector_length	Longest CSS selector, in bytes. The selection functions take no `limits`: they use the limits the document was parsed with. It bounds the selectors you pass, not the ones the extraction functions such as `html_forms()` use internally.

Details

The memory limit is the real guard; the input limit is a cheap check before parsing starts. Parsing and conversion together need about 26 bytes of native memory per input byte on ordinary markup, and 40 to 90 on markup dense with small elements, tables or formatting, so a large enough page reaches `max_memory` before `max_input` or `max_nodes`: a `zuhtml_limit_error`, not a crash. At the defaults, 16 MiB of ordinary markup (about 1.5 million nodes, 450 MB) parses.

Value

An object of class `zuhtml_limits`: a named list of the limits, as doubles.

See Also

Other parsing: `html_fragment()`, `html_info()`, `html_parse()`, `html_problems()`, `zuhtml_info()`

Examples

```
html_limits()

# A tighter nesting limit for a service parsing untrusted pages:
lim <- html_limits(max_depth = 128)
try(html_parse(strrep("<div>", 200), limits = lim))
```

html_links *Links in a document*

Description

Finds the <a> and <area> elements with an href below the given nodes (and the nodes themselves), in document order, without duplicates. Duplicate destinations are kept, and so are fragment, mailto: and tel: links.

Usage

```
html_links(x, absolute = FALSE)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
absolute	If TRUE, url is resolved with html_url(); otherwise it is href unchanged.

Value

A data frame with one row per link and character columns text (the link's cleaned text, see html_text_clean()), href (the attribute as written, decoded) and url.

See Also

Other extraction: html_forms(), html_list(), html_table(), html_table_cells(), html_url()

Examples

```
doc <- html_parse(
  "<nav><a href='/'>Home</a> <a href='about.html'>About</a></nav>",
  base_url = "https://example.org/site/"
)
html_links(doc)
html_links(doc, absolute = TRUE)
```

html_list *Extract an HTML list*

Description

Reads one <ul> or <ol> element. In "text" mode, the result has one string per item: the item's cleaned text (see html_text_clean()), without the text of any list nested inside it, so child items do not leak into their parent. A nested list still separates the text around it with a line break. In "tree" mode, nested lists become children of the item that contains them, including lists inside wrapper elements such as .

Usage

```
html_list(x, mode = c("text", "tree"))
```

Arguments

x	A zuhtml_nodeset holding exactly one or element, as from html_element().
mode	"text" or "tree".

Details

Items are the <li> children of the list, in source order; empty items are "" and duplicates are kept.

Value

- "text": a character vector, one string per item.
- "tree": an object of class zuhtml_list, a list with type ("ul" or "ol") and items, a list with one element per item; each item is a list with text (as in text mode) and children (a list of zuhtml_list objects, one per list nested in the item).

See Also

Other extraction: html_forms(), html_links(), html_table(), html_table_cells(), html_url()

Examples

```
doc <- html_parse("<ul><li>Apples<li>Tools<ul><li>Hammer<li>Saw</ul></ul>")
items <- html_element(doc, "ul")
html_list(items)
html_list(items, mode = "tree")
```

---

html_markdown *Convert HTML to Markdown*

Description

Writes each node as CommonMark text, for reading, for notes, or as input to a language model. Text is skipped and collapsed as html_text_clean() does, and Markdown-significant characters in it are escaped, so the text reads back as written.

Usage

```
html_markdown(x)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.

Details

The conversion covers:

- headings (# to ######), paragraphs and the other block elements (separated by blank lines), and thematic breaks (<hr>, as ---, or as *** where it opens a - list item);
- emphasis (, <i>) as *...* and strong (, <b>) as **...**, where CommonMark reads it back as written: emphasis that cannot be written safely, such as emphasis inside emphasis of its kind, or bold italics of one element (written as **...**), keeps its text without the markup;
- inline code (<code>, <kbd>, <samp>, <tt>) as a code span, with whitespace at its ends written outside it; adjacent spans join into one;
- <pre> as a fenced code block, with its language when a language-* or lang-* class names one;
- block quotes, and ordered and unordered lists, nested, with start, reversed and value honoured; empty list items are left out and take no number. A list right after another of its kind takes the other bullet (- or *) or delimiter (. or)), so that the two stay separate;
- links and images, their URLs resolved against the document's base URL as html_url() does, or as written where that is not possible; <a> without href is plain text, and <img> without src is left out;
-
 as a hard line break;
- data tables as GFM pipe tables: tables in which no cell spans rows or columns and every cell holds only inline content (no paragraphs, lists, headings, s, code blocks or tables). Rows are in html_table()'s order (head, bodies, foot), the first is the header row, and cell contents stay on one line. Other tables, such as tables used for page layout, are written as their content, one paragraph per row, with cells separated by spaces.

<head>, <script>, <style>, <template>, <iframe>, <svg> and comments are left out. Other elements contribute their text only. Markup that crosses block boundaries, such as <b> around two paragraphs, is closed and reopened in each block.

Value

A character vector as long as x: the Markdown of each element, document, fragment or text node, without a trailing newline; NA for other nodes and for missing nodes.

See Also

html_text_clean() for plain text.

Other node values: html_attr(), html_name(), html_serialize(), html_strings(), html_text(), html_text_clean()

Examples

```
doc <- html_parse(paste0(
  "<h1>Release notes</h1>",
  "<p>Version <b>2.0</b> adds <code>fetch()</code>. ",
  "See <a href='changes.html'>the changes</a>.</p>",
  "<ul><li>Faster parsing</li><li>New <i>options</i>:",
```

```
  "<ol><li>limits</li><li>encoding</li></ol></li></ul>",
  "<table><tr><th>Item</th><th>Cost</th></tr>",
  "<tr><td>Tea</td><td>3</td></tr></table>"
), base_url = "https://example.org/docs/")
cat(html_markdown(doc))
```

`html_meta` *Meta tags*

Description

One row per `<meta>` element below the given nodes (and the nodes themselves), in document order. Duplicates are kept, so OpenGraph (`og:*` in `property`), Twitter cards and Dublin Core (`name`) come out as rows to filter.

Usage

```
html_meta(x)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.

Value

A data frame with character columns `name`, `property`, `http_equiv`, `charset` and `content`, each the attribute as written (decoded) or `NA` when absent. `name`, `property` and `http_equiv` keep their case; compare with `tolower()`.

See Also

Other metadata: `html_json_ld()`, `html_microdata()`, `html_title()`

Examples

```
doc <- html_parse(paste0(
  "<meta charset='utf-8'>",
  "<meta name='description' content='A page.'>",
  "<meta property='og:title' content='Title'>",
  "<meta property='og:image' content='a.png'>",
  "<meta property='og:image' content='b.png'>"
))
meta <- html_meta(doc)
meta
meta$content[meta$property %in% "og:image"]
```

`html_microdata` *Microdata items*

Description

The top-level microdata items (elements with `itemscope` and no `itemprop`) below the given nodes (and the nodes themselves), in document order, with their properties collected by the HTML standard's algorithm (https://html.spec.whatwg.org/multipage/microdata.html), including properties pulled in by `itemref`. RDFa is not read.

Usage

```
html_microdata(x)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.

Details

Each property's value follows the standard: a nested item for an element with `itemscope`; the `content` attribute of `<meta>`; the resolved `src` of `<audio>`, `<embed>`, `<iframe>`, `<img>`, `<source>`, `<track>` and `<video>`, `href` of `<a>`, `<area>` and `<link>`, and `data` of `<object>` (resolved as `html_url()` does, `""` when the attribute is absent or does not resolve); the `value` attribute of `<data>` and `<meter>`; the `datetime` attribute of `<time>` when present. Otherwise it is the element's text, which here is cleaned as `html_text_clean()` does, not the raw `textContent`. An item that is its own ancestor through `itemref` is `NULL`.

Value

A list of items. Each item is a list with `type` (the tokens of `itemtype`, a character vector, possibly empty), `id` (`itemid` resolved as a URL, `NA` when absent or without `itemtype`) and `properties`: a named list, in document order of first appearance, of lists of values, since a name may occur more than once. A value is a string or a nested item.

See Also

Other metadata: `html_json_ld()`, `html_meta()`, `html_title()`

Examples

```
doc <- html_parse(paste0(
  "<div itemscope itemtype='https://schema.org/Book'>",
  "<span itemprop='name'>Dune</span>",
  "<div itemprop='author' itemscope itemtype='https://schema.org/Person'>",
  "<span itemprop='name'>Frank Herbert</span></div>",
  "<meta itemprop='isbn' content='9780441013593'>",
  "</div>"
```

```
))
book <- html_microdata(doc)[[1]]
book$type
book$properties$name[[1]]
book$properties$author[[1]]$properties$name[[1]]
```

`html_name` *Node names, namespaces and types*

Description

Vectorized accessors returning one value per node, with `NA` for missing nodes.

Usage

```
html_name(x)

html_namespace(x)

html_type(x)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.

Value

A character vector as long as `x`.

- `html_name()`: the element name, lowercase for HTML elements and in the spec's mixed case for SVG ones such as `"foreignObject"`; the doctype's name for a doctype node; `NA` for other nodes.
- `html_namespace()`: the namespace URI of an element, for example `"http://www.w3.org/1999/xhtml"` or `"http://www.w3.org/2000/svg"`; `NA` for other nodes.
- `html_type()`: one of `"document"`, `"fragment"`, `"doctype"`, `"element"`, `"text"`, `"comment"` and `"processing_instruction"`. A `<template>` is an `"element"`.

See Also

Other node values: `html_attr()`, `html_markdown()`, `html_serialize()`, `html_strings()`, `html_text()`, `html_text_clean()`

Examples

```
doc <- html_parse("<p>Text<svg><foreignObject/></svg><!-- note -->")
nodes <- html_children(html_children(html_root(doc))[2], FALSE)
nodes <- html_children(nodes, FALSE)
html_type(nodes)
html_name(nodes)
html_namespace(nodes)
```

`html_parse` *Parse HTML*

Description

`html_parse()` parses one string or raw vector of HTML the way a browser does: omitted end tags, unquoted attributes, character references and misnested elements are repaired by the HTML parsing algorithm, never rejected. `html_read()` reads and parses one file, URL or connection.

Usage

```
html_parse(
  x,
  encoding = NULL,
  base_url = NULL,
  comments = TRUE,
  limits = html_limits()
)

html_read(
  path,
  encoding = NULL,
  base_url = NULL,
  comments = TRUE,
  limits = html_limits()
)
```

Arguments

`x`	One string, or a raw vector of encoded bytes.
`encoding`	The encoding of raw input: UTF-16 or the name of an ASCII-compatible encoding `iconv()` accepts (see "Encoding"); `NULL` to use a byte-order mark, the page's declaration, or UTF-8.
`base_url`	The document's URL, used to resolve relative links; `NULL` if unknown.
`comments`	Whether to keep comment nodes.
`limits`	Resource limits from `html_limits()`.
`path`	One file path, URL or connection; see "Reading files, URLs and connections".

Details

`html_parse()` never treats a string as a file name or URL.

Value

A `zuhtml_document`.

Reading files, URLs and connections

`html_read()` reads its input as raw bytes, then decodes and parses them as `html_parse()` does:

- a string with an `http`, `https`, `ftp`, `ftps` or `file` scheme (in any case) is read with `url()`, and becomes the document's `base_url` unless `base_url` is given. A redirect is not seen, so after one the base URL is the address asked for. HTTP headers are not read either: the encoding comes from a byte-order mark, `encoding` or the page's `<meta>`. For anything more (headers, authentication, retries), fetch with an HTTP client and pass the body to `html_parse()`;
- any other string is a file path;
- a connection, such as `gzfile()` or `rawConnection()`, is read as `readBin()` reads one: an unopened connection is opened in `"rb"` mode and closed afterwards; an open one must be in binary mode, is read from its current position, and is left open. A non-blocking pipe or socket that has no data yet is an error rather than a short read.

Reading stops with a `zuhtml_limit_error` as soon as the input passes four times `max_input` bytes, before a larger file is read at all. A file that does not exist, an unreachable URL, and a connection that cannot be read are `zuhtml_input_errors`. The input is always read whole before parsing, because decoding needs all of it. The other arguments mean what they mean to `html_parse()`, whatever `path` is.

Encoding

A character string is taken as text: it is converted to UTF-8 with `enc2utf8()`, and `encoding` must be `NULL` or `"UTF-8"`. In a C locale, where `enc2utf8()` cannot convert, an unmarked string's non-ASCII bytes are read as UTF-8, as `readLines()` returns a UTF-8 file's lines there. A string marked as `"bytes"` is rejected; pass a raw vector instead.

A raw vector is decoded with, in order of precedence, a byte-order mark (UTF-8, UTF-16LE or UTF-16BE), `encoding`, a declaration in the page, or UTF-8. A byte-order mark that contradicts `encoding` is an error, and so is any byte sequence that is invalid in the chosen encoding: nothing is replaced silently. A fetcher that knows the HTTP charset should pass it as `encoding`.

Raw input longer than four times `max_input` is refused before it is decoded, whatever it decodes to: no supported encoding needs more than four bytes for a character, but the escape sequences of a stateful encoding such as ISO-2022-JP decode to nothing.

UTF-16 is decoded by zuhtml itself, and windows-1255 and windows-1258 from the Encoding Standard's indexes, so that a combining mark stays a character of its own as in a browser. Every other encoding is converted by `iconv()` and must be ASCII-compatible: UCS-2, UTF-32 and EBCDIC are encoding errors.

The declaration is found as browsers find it, by the HTML standard's prescan of the first 1024 bytes for `<meta charset="...">` or `<meta http-equiv="Content-Type" content="...; charset=...">`. The prescan skips comments and the insides of tags, but not the text of scripts. Labels are those of the Encoding Standard, which maps several to a superset: `"iso-8859-1"`, `"latin1"` and `"us-ascii"` mean windows-1252, `"gb2312"` means GBK, and a UTF-16 label means UTF-8 (the bytes read as ASCII, so they are not UTF-16). An unknown label is ignored. `html_info()` reports the encoding used and its source.

A file saved in another encoding without updating its declaration, as some tools do when they convert pages to UTF-8, decodes wrongly or fails to decode, as it would in a browser. Pass its real encoding as `encoding`.

A leading byte-order mark is removed. Input containing a NUL character after decoding is rejected.

See Also

`html_problems()` for the parse errors that were repaired; `html_limits()`; zuhtml-conditions for the errors these functions raise.

Other parsing: `html_fragment()`, `html_info()`, `html_limits()`, `html_problems()`, `zuhtml_info()`

Examples

```
doc <- html_parse("<p>Hello <b>world</b>")
doc

# Raw bytes in a declared encoding:
html_parse(as.raw(c(0x3c, 0x70, 0x3e, 0xe9)), encoding = "latin1")

# From a file:
path <- tempfile(fileext = ".html")
writeLines("<title>Saved page</title><p>Text", path)
html_read(path)

# From a compressed file, through a connection:
gz <- tempfile(fileext = ".html.gz")
writeLines("<p>Compressed", gzfile(gz))
html_read(gzfile(gz))

# From a URL, when online; relative links resolve against it:
if (interactive()) {
  doc <- html_read("https://cran.r-project.org/web/packages/")
  head(html_links(doc, absolute = TRUE))
}
```

---

html_problems *Parse problems recorded for a document*

Description

HTML parsing never fails on malformed markup: the parser repairs it as a browser would. `html_problems()` lists what was repaired, which is useful for diagnosing why a tree looks the way it does. Real pages nearly always have some. This is not a conformance validator.

Usage

```
html_problems(x)
```

Arguments

x	A `zuhtml_document`, or a `zuhtml_nodeset` for the document that owns it.

Value

A data frame with one row per problem, in input order, and columns `stage` (`"tokenizer"` or `"parser"`), `code` (a stable package-owned name, such as `"unexpected-end-tag"` or `"duplicate-attr"`), `line` and `column` (1-based) and `byte_offset` (0-based, into the decoded UTF-8 input). At most `max_errors` problems are kept (see `html_limits()`); attribute `"truncated"` is `TRUE` when more occurred.

See Also

Other parsing: `html_fragment()`, `html_info()`, `html_limits()`, `html_parse()`, `zuhtml_info()`

Examples

```
doc <- html_parse("<p>One</div><p id=a id=b>Two")
html_problems(doc)
```

---

html_serialize *Serialize nodes as HTML*

Description

Produces normalized HTML following the WHATWG serialization algorithm, not the original input bytes: end tags the input left out are written, attribute values are always double-quoted, and &, <, >, " and non-breaking spaces are escaped as the algorithm requires. Parsing the result gives back the same tree, except where the HTML standard itself does not guarantee that (for example markup that tree construction rearranges).

Usage

```
html_serialize(x, outer = TRUE, pretty = FALSE)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.
`outer`	If `TRUE`, each node with its own markup, like `outerHTML`; if `FALSE`, only its contents, like `innerHTML`. A document or fragment has no markup of its own, so both give its contents.
`pretty`	If `TRUE`, lay block-level elements out on their own lines, indented two spaces per level, for reading. Inline content stays on its line, whitespace-only text between blocks is dropped, and the contents of `<pre>`, `<textarea>`, `<script>` and `<style>` are left exactly as they are. The result is for people, not for parsing again: the added whitespace becomes text.

Details

Serialization is not sanitization: `<script>` elements and `javascript:` URLs are written as they were parsed.

`as.character()` on a nodeset is `html_serialize()`.

Value

A character vector as long as x; NA for missing nodes.

See Also

Other node values: html_attr(), html_markdown(), html_name(), html_strings(), html_text(), html_text_clean()

Examples

```
doc <- html_parse("<p class=x>One<br>Two & <b>three</p>")
html_serialize(doc)
p <- html_children(html_children(html_root(doc))[2])
html_serialize(p)
html_serialize(p, outer = FALSE)
cat(html_serialize(doc, pretty = TRUE))

# To write a document to a file:
path <- tempfile(fileext = ".html")
writeLines(html_serialize(doc), path)
```

---

html_strings *Text pieces of nodes*

Description

The text nodes of each node's subtree, in tree order, as separate strings: the pieces html_text() concatenates. Boundaries between elements are kept, which matters when the markup, not whitespace, separates values (<td>1</td><td>2</td> is "1", "2", not "12"). Text inside <script> and <style> (HTML's and SVG's) and <template> is skipped, as are comments.

Usage

```
html_strings(x, trim = FALSE, drop_empty = FALSE)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
trim	If TRUE, remove leading and trailing whitespace from each piece.
drop_empty	If TRUE, drop pieces that are empty (after trimming, when trim = TRUE).

Value

A list as long as x of character vectors; NA_character_ for a missing node. A text node is its own single piece.

See Also

Other node values: html_attr(), html_markdown(), html_name(), html_serialize(), html_text(), html_text_clean()

Examples

```
doc <- html_parse("<p>One <b>two</b>\n  <i> three </i></p>")
p <- html_element(doc, "p")
html_strings(p)
html_strings(p, trim = TRUE, drop_empty = TRUE)
```

---

html_table *Extract HTML tables as data frames*

Description

html_table() reads one <table> element into a data frame of character columns. html_tables() finds tables and reads each.

Usage

```
html_table(
  x,
  header = "auto",
  trim = TRUE,
  na = character(),
  convert = FALSE,
  decimal = ".",
  thousands = NULL,
  limits = html_limits()
)

html_tables(x, css = "table", match = NULL, ...)
```

Arguments

x	For html_table(), a nodeset holding exactly one <table> element; for html_tables(), a zuhtml_document or zuhtml_nodeset to search.
header	How to find header rows; see the Headers section.
trim	Whether to trim each cell's text; see html_text_clean().
na	Strings that become NA after trimming, such as c("", "-"). By default no text is treated as missing.
convert	If TRUE, convert columns that are entirely numbers or logicals; see the Conversion section.
decimal	The decimal mark for convert: a single character.

thousands	The grouping mark for convert: a single character other than decimal, or NULL for none.
limits	Resource limits from html_limits(); max_table_cells bounds the grid, spans included.
css	For html_tables(), the selector of tables to read. Only the outermost matching tables are read; nested ones remain part of their parents' cells.
match	For html_tables(), a regular expression (grepl()) that a table's cleaned text must match for the table to be read, or NULL to read every table.
...	For html_tables(), arguments passed on to html_table().

Value

html_table(): a data frame of character columns. A table with no rows gives a data frame with no rows and no columns; a table with only header rows gives no rows and the named columns. html_tables(): a list of such data frames, list() when there are none.

The grid

Rows are read in logical order – every <thead>, then the <tbody> sections (and any rows directly in the table), then every <tfoot> – and footer rows are ordinary data rows. Rows of nested tables never become rows of the outer table, and a nested table's text is left out of the cell that holds it (it still separates the text around it with a line break).

Each cell is placed at the next free column of its row. colspan and rowspan are read with HTML's rules for non-negative integers: a missing or invalid value is 1, colspan="0" is 1, and values are capped at 1000 columns and 65534 rows. rowspan="0" extends to the end of the cell's row group, and a larger rowspan is cut silently at the end of its group. A spanned value is repeated in every slot it covers. A cell whose span would cover a slot another cell already covers is an error of class zuhtml_table_structure_error, never a silent overwrite. Slots no cell covers, in ragged rows, are NA; an empty cell is "".

Headers

- "auto": the rows of <thead> sections if there are any, otherwise the leading rows whose cells are all <th> (a <th> among <td>s does not make a header row);
- TRUE: the first row; FALSE: none;
- a vector of row numbers: those rows.

Header rows are removed from the data. A column's name joins the non-empty header texts above it with " / ", merging repeats that a colspan created. Blank names become V1, V2, ... by column position, and duplicates are made unique with make.unique(). Names are not otherwise made syntactic.

Cell text follows html_text_clean(). All columns are character unless convert = TRUE.

Conversion

With convert = TRUE, each column is converted on its own, and only when every non-missing value converts; otherwise it stays character, so conversion never turns a value into NA. Values are compared after removing surrounding whitespace.

- TRUE, FALSE, true, false, True and False make a logical column.
- Decimal numbers, with an optional sign and exponent, make an integer column when all are whole and fit, otherwise a double column. decimal is the decimal mark. thousands, if given, is a grouping mark allowed between groups of three digits, as in 1,234,567.5; a badly grouped value such as 1,23 keeps the column character. Cell text has its non-breaking spaces turned into spaces, so thousands = "\u00a0" matches either.
- A value with a leading zero, such as "0012", or "0,123" with thousands = ",", keeps the column character: it is an identifier, not a number. "0" and "0.5" are numbers.
- A value with more than 15 significant digits, which a double need not hold exactly, or one that overflows to infinity, underflows to zero or is subnormal, such as "12345678901234567890", "1e999" or "1e-310", keeps the column character.
- Inf, NaN, NA, currency symbols and percentages are not numbers; list such strings in na, or convert them yourself. A column with no non-missing values stays character.

See Also

Other extraction: html_forms(), html_links(), html_list(), html_table_cells(), html_url()

Examples

```
doc <- html_parse(paste0(
  "<table><thead><tr><th>Product<th>Price</thead>",
  "<tbody><tr><td>0012<td>12.50<tr><td>0034<td>9.00</tbody></table>"
))
html_table(html_element(doc, "table"))

doc <- html_parse(paste0(
  "<table><tr><th rowspan=2>Region<th colspan=2>Sales",
  "<tr><th>2025<th>2026<tr><td>North<td>1<td>2</table>"
))
html_tables(doc)
html_tables(doc, match = "North")

doc <- html_parse(paste0(
  "<table><tr><th>Id<th>Amount<th>Paid",
  "<tr><td>007<td>1.234,50<td>true<tr><td>012<td>99<td>false</table>"
))
str(html_table(html_element(doc, "table"), convert = TRUE,
               decimal = ",", thousands = "."))
```

---

html_table_cells *Cells of an HTML table*

Description

One row per cell of one <table>, with its position in the grid that html_table() builds: spans are placed, clipped and checked exactly as there, and rows are numbered in the same logical order, header rows included. Use it when the data frame loses what you need: which cells are headers, how far they span, or the links inside them.

Usage

```
html_table_cells(x, trim = TRUE, absolute = FALSE, limits = html_limits())
```

Arguments

x	A nodeset holding exactly one <table> element.
trim	Whether to trim each cell's text; see html_text_clean().
absolute	If TRUE, links are resolved with html_url(); otherwise they are href as written, as in html_links().
limits	Resource limits from html_limits(); max_table_cells bounds the grid, spans included.

Value

A data frame with one row per <td> or <th>, in grid order (by row, then column), and columns:

- row, column: the integer position of the cell's top-left slot;
- rowspan, colspan: the integer number of rows and columns it covers, after rowspan="0" is expanded and spans are clipped;
- section: "thead", "tbody" or "tfoot"; rows directly in the table are "tbody";
- header: TRUE for a <th>;
- text: the cell's cleaned text, without nested tables, as in html_table();
- links: a list of character vectors, the href of each <a> and <area> in the cell, outside nested tables.

See Also

Other extraction: html_forms(), html_links(), html_list(), html_table(), html_url()

Examples

```
doc <- html_parse(paste0(
  "<table><tr><th rowspan=2>Package<th colspan=2>Links",
  "<tr><th>Home<th>Docs",
  "<tr><td>zuhtml<td><a href='https://example.org/'>site</a>",
  "<td><a href='/ref/'>ref</a> <a href='/news/'>news</a></table>"
), base_url = "https://example.org/pkg/")
cells <- html_table_cells(html_element(doc, "table"), absolute = TRUE)
cells[, c("row", "column", "rowspan", "colspan", "header", "text")]
cells$links[[7]]
```

html_text *Text content of nodes*

Description

`html_text()` is the structural text of each node: the text nodes in its subtree, concatenated in tree order, exactly as parsed. It inserts no separators, trims nothing and keeps `<script>` and `<style>` text; it does not include comments or the contents of `<template>` elements.

Usage

```
html_text(x, recursive = TRUE)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.
`recursive`	If `FALSE`, only the direct text children of each node.

Value

A character vector as long as `x`. An element with no text is `""`; a text, comment or processing-instruction node is its own content; a doctype and a missing node are `NA`.

See Also

Other node values: `html_attr()`, `html_markdown()`, `html_name()`, `html_serialize()`, `html_strings()`, `html_text_clean()`

Examples

```
doc <- html_parse("<p>Hello <b>big</b> world</p>")
p <- html_children(html_children(html_root(doc))[2])
html_text(p)
html_text(p, recursive = FALSE)
```

html_text_clean *Cleaned text for extraction*

Description

Text as a reader would want it from a page, rather than exactly as parsed. It is not a browser's `innerText`: there is no layout or CSS, and hidden elements are included. The rules are fixed and documented:

- text inside `<script>`, `<style>` (HTML's or SVG's) and `<template>` is skipped, and so are comments;
- runs of whitespace (space, tab, newline, carriage return, form feed) collapse to a single space, except inside `<pre>`, `<textarea>`, `<listing>` and `<plaintext>`, where whitespace is kept;
- `<br>` is a line break; each block element is a line break before and after, and adjacent block boundaries give a single line break. The block elements are `address`, `article`, `aside`, `blockquote`, `body`, `caption`, `center`, `dd`, `details`, `dialog`, `dir`, `div`, `dl`, `dt`, `fieldset`, `figcaption`, `figure`, `footer`, `form`, `h1` to `h6`, `head`, `header`, `hgroup`, `hr`, `html`, `legend`, `li`, `listing`, `main`, `menu`, `nav`, `ol`, `optgroup`, `option`, `p`, `plaintext`, `pre`, `section`, `summary`, `table`, `tbody`, `tfoot`, `thead`, `title`, `tr`, `ul` and `xmp`;
- table cells (`<td>`, `<th>`) are separated by a space.

Usage

```
html_text_clean(x, trim = TRUE, nbsp = TRUE)
```

Arguments

`x`	A `zuhtml_document` or `zuhtml_nodeset`.
`trim`	If `TRUE`, no whitespace at the start or end, even from `<pre>` or `<textarea>`. If `FALSE`, whitespace there is kept, collapsed to a single space outside preformatted elements.
`nbsp`	If `TRUE`, treat non-breaking spaces (U+00A0) as whitespace, so they collapse like spaces; inside `<pre>` they become spaces.

Value

A character vector as long as `x`: the cleaned text of each element, document, fragment or text node; `NA` for other nodes and for missing nodes.

See Also

`html_text()` for the text exactly as parsed.

Other node values: `html_attr()`, `html_markdown()`, `html_name()`, `html_serialize()`, `html_strings()`, `html_text()`

Examples

```
doc <- html_parse(paste0(
  "<div><h1>Title</h1>\n  <p>Some   <b>bold</b> text.<br>New line",
  "<script>ignored()</script></p><pre>  kept\n  as is</pre></div>"
))
div <- html_element(doc, "div")
```

```
cat(html_text_clean(div))
html_text(div)
```

`html_title` *Document title*

Description

The document's title as the HTML standard defines `document.title`: the text of the first `<title>` element in the HTML namespace, with leading and trailing whitespace removed and runs of whitespace collapsed to one space. An SVG `<title>` does not count.

Usage

```
html_title(x)
```

Arguments

`x`	A `zuhtml_document`, or a `zuhtml_nodeset` whose document is used.

Value

A single string, `NA` when the document has no `<title>`.

See Also

Other metadata: `html_json_ld()`, `html_meta()`, `html_microdata()`

Examples

```
html_title(html_parse("<title>  Annual\n  report </title><p>Body"))
html_title(html_parse("<p>No title"))
```

`html_url` *Resolve URLs in attributes*

Description

Reads a URL-valued attribute from each node and resolves it against the document's base URL with the reference-resolution algorithm of RFC 3986, section 5 (https://www.rfc-editor.org/rfc/rfc3986.html#section-5): dot segments are removed and relative, query-only, fragment-only and protocol-relative references are handled. Nothing is fetched.

Usage

```
html_url(x, attr = "href", base_url = NULL)
```

Arguments

x	A zuhtml_document or zuhtml_nodeset.
attr	The attribute holding the URL.
base_url	The base URL to resolve against, overriding the document's; NULL to use the document's.

Details

The base URL is base_url if given; otherwise the document's first HTML <base href> (not an SVG or MathML <base>) resolved against the base_url given to html_parse(); otherwise that base_url alone.

This is RFC 3986, not the WHATWG URL Standard browsers implement: there is no IDNA processing, no percent-encoding of characters that need it, and no special handling of backslashes. Leading and trailing ASCII whitespace is removed from the attribute; a reference that still contains whitespace or a control character, or whose scheme is invalid, is malformed.

Value

A character vector as long as x: the resolved URL of each node. NA where the node lacks the attribute, the reference is malformed, or a relative reference has no absolute base URL to resolve against; and for missing nodes. The original attribute value stays available through html_attr().

See Also

Other extraction: html_forms(), html_links(), html_list(), html_table(), html_table_cells()

Examples

```
doc <- html_parse(
  "<a href='../img/a.png'>A</a><a href='//cdn.example.org/b'>B</a>",
  base_url = "https://example.org/docs/page.html"
)
html_url(html_elements(doc, "a"))
html_url(html_elements(doc, "a"), base_url = "http://other.test/x/")
```

---

zuhtml-conditions *Conditions raised by zuhtml*

Description

Every error zuhtml raises is a condition of class zuhtml_error with one more specific subclass. Handle them by class, for example with tryCatch(..., zuhtml_limit_error = function(e) ...), never by matching the message text, which may change.

Details

- zuhtml_input_error: an argument is invalid, for example x is not a single string or raw vector, a file does not exist, a URL or connection cannot be opened or read, or a limit is not a whole number in range. Field arg names the argument.
- zuhtml_encoding_error: raw input cannot be decoded: an unknown encoding, a byte sequence invalid in the declared encoding, or a byte-order mark that contradicts encoding. Field encoding.
- zuhtml_limit_error: a resource limit from html_limits() was exceeded. Fields limit (its name, such as "max_depth"), maximum (its value) and observed (the value that tripped it, where known). Nothing is returned and all native memory is released. Three fixed limits use the same class: "arena", the most attributes or string-pool bytes one document can hold (just under 2^{32}); "string_length", R's limit on the bytes of one string; and "memory", when a result being built grows past what a size can count. Results (text, Markdown, serialized HTML, table grids) are built in memory R allocates, so running out of memory there is not a zuhtml_error: it is R's own error, such as "cannot allocate vector of size ...", and R releases what was allocated. Only parsing has a native memory budget, max_memory.
- zuhtml_parse_error: an internal invariant of the parser failed. This is a bug; please report it.
- zuhtml_pointer_error: a document is no longer available, for example because it was restored from a saved R session. Parse the HTML again.

Recoverable HTML errors, such as a missing end tag, are not conditions: they are repaired as a browser would repair them and recorded for html_problems().

---

zuhtml_info *Report the zuhtml build configuration*

Description

Reports the bundled 'Gumbo' version and the local patches applied to it, and runs two self-tests against the compiled parser. Intended for diagnostics and bug reports: in a correct build both parser_ok and depth_limit_ok are TRUE.

Usage

```
zuhtml_info()
```

Value

An object of class zuhtml_info: a list with elements zuhtml_version, gumbo_version, gumbo_patches (a character vector of patch identifiers, in the order applied), parser_ok (the bundled parser builds the expected tree for a fixed document) and depth_limit_ok (the parse-time nesting limit stops a deeply nested document).

See Also

Other parsing: html_fragment(), html_info(), html_limits(), html_parse(), html_problems()

Examples

```
zuhtml_info()
```

Index

∗ **extraction**
  html_forms, 8
  html_links, 14
  html_list, 14
  html_table, 25
  html_table_cells, 27
  html_url, 31
∗ **metadata**
  html_json_ld, 11
  html_meta, 17
  html_microdata, 18
  html_title, 31
∗ **navigation**
  html_children, 4
  html_closest, 5
  html_elements, 6
∗ **node values**
  html_attr, 2
  html_markdown, 15
  html_name, 19
  html_serialize, 23
  html_strings, 24
  html_text, 29
  html_text_clean, 29
∗ **parsing**
  html_fragment, 9
  html_info, 10
  html_limits, 12
  html_parse, 20
  html_problems, 22
  zuhtml_info, 33

enc2utf8(), *21*

grepl(), *26*
gzfile(), *21*

html_ancestors *(*html_children*)*, 4
html_attr, 2
html_attr(), *16*, *19*, *24, 25*, *29, 30*, *32*
html_attrs *(*html_attr*)*, 2
html_children, 4
html_children(), *6, 7*
html_classes *(*html_attr*)*, 2
html_closest, 5
html_closest(), *5*, *7*
html_document *(*html_children*)*, 4
html_element *(*html_elements*)*, 6
html_element(), *15*
html_elements, 6
html_elements(), *5, 6*
html_filter *(*html_elements*)*, 6
html_forms, 8
html_forms(), *13–15*, *27, 28*, *32*
html_fragment, 9
html_fragment(), *11*, *13*, *22, 23*, *33*
html_info, 10
html_info(), *10*, *13*, *21–23*, *33*
html_json_ld, 11
html_json_ld(), *17, 18*, *31*
html_limits, 12
html_limits(), *6, 7*, *10, 11*, *20*, *22, 23*, *26*, *28*, *33*
html_links, 14
html_links(), *9*, *15*, *27, 28*, *32*
html_list, 14
html_list(), *9*, *14*, *27, 28*, *32*
html_markdown, 15
html_markdown(), *3*, *19*, *24, 25*, *29, 30*
html_matches *(*html_elements*)*, 6
html_meta, 17
html_meta(), *12*, *18*, *31*
html_microdata, 18
html_microdata(), *12*, *17*, *31*
html_name, 19
html_name(), *3*, *16*, *24, 25*, *29, 30*
html_namespace *(*html_name*)*, 19
html_next_sibling *(*html_children*)*, 4
html_parent *(*html_children*)*, 4

html_parse, 20
html_parse(), *10–13*, *23*, *32, 33*
html_previous_sibling *(*html_children*)*, 4
html_problems, 22
html_problems(), *10, 11*, *13*, *22*, *33*
html_read *(*html_parse*)*, 20
html_root *(*html_children*)*, 4
html_root(), *10*
html_serialize, 23
html_serialize(), *3*, *16*, *19*, *25*, *29, 30*
html_strings, 24
html_strings(), *3*, *16*, *19*, *24*, *29, 30*
html_table, 25
html_table(), *9*, *14–16*, *27, 28*, *32*
html_table_cells, 27
html_table_cells(), *9*, *14, 15*, *27*, *32*
html_tables *(*html_table*)*, 25
html_template_content *(*html_children*)*, 4
html_text, 29
html_text(), *3*, *16*, *19*, *24, 25*, *30*
html_text_clean, 29
html_text_clean(), *3*, *14–16*, *18, 19*, *24–26*, *28, 29*
html_title, 31
html_title(), *12*, *17, 18*
html_type *(*html_name*)*, 19
html_type(), *10*
html_url, 31
html_url(), *9*, *14–16*, *18*, *27, 28*

iconv(), *20, 21*

jsonlite::parse_json(), *12*

make.unique(), *26*

rawConnection(), *21*
readBin(), *21*
readLines(), *21*

tolower(), *17*

url(), *21*

zuhtml-conditions, *12*, *22*, 32
zuhtml_info, 33
zuhtml_info(), *10, 11*, *13*, *22, 23*