

# Package ‘unnest’

May 8, 2026

**Title** Unnest Hierarchical Data Structures

**Version** 0.1

**Description** Fast flattening of hierarchical data structures (e.g. JSON, XML) into data.frames with a flexible spec language.

**License** GPL (>= 2)

**Encoding** UTF-8

**RoxygenNote** 7.3.1

**Suggests** data.table, dplyr, knitr, repurrrsive, rmarkdown, testthat, tibble, tidyr

**VignetteBuilder** knitr

**URL** <https://github.com/vspinu/unnest/>,  
<https://vspinu.github.io/unnest/>

**BugReports** <https://github.com/vspinu/unnest/issues>

**NeedsCompilation** yes

**Author** Vitalie Spinu [aut, cre]

**Maintainer** Vitalie Spinu <spinuvit@gmail.com>

**Repository** CRAN

**Date/Publication** 2025-11-23 19:10:03 UTC

## Contents

spec . . . . .	2
unnest . . . . .	3

<b>Index</b>	<b>7</b>
--------------	----------

spec

*Unnest spec***Description**

Unnest spec is a nested list with the same structure as the nested json. It specifies how the deeply nested lists ought to be unnested. `spec()` is a handy constructor for spec lists. `s()` is a shorthand alias for `spec()`.

**Usage**

```
spec(
  selector = NULL,
  ...,
  as = NULL,
  children = NULL,
  groups = NULL,
  include = NULL,
  exclude = NULL,
  stack = NULL,
  process = NULL,
  default = NULL
)
```

```
s(
  selector = NULL,
  ...,
  as = NULL,
  children = NULL,
  groups = NULL,
  include = NULL,
  exclude = NULL,
  stack = NULL,
  process = NULL,
  default = NULL
)
```

**Arguments**

`selector` A shorthand syntax for an include parameter. Can be a list or a character vector.

1. When `selector` is a list or a character vector with length greater than 1, each element is an include parameter at the corresponding level. For example `s(c("a", "b"), ...)` is equivalent to `s(include = "a", s(include = "b", ...))`
2. When `selector` is a character of length 1 and contains "/" characters it is split with "/" first. For instance `s(c("a", "b"), ...)`, `s("a/b", ...)`

and `s("a", s("b", ...))` are all equivalent to the canonical `s(include = "a", s(include = "b", ...))`. Components consisting entirely of digits are converted to integers. For example `s("a/2/b" ...)` is equivalent to `s("a", s(2, s("b", ...)))`

- Multiple `include` fields can be separated with `,`. For example `s("a/b,c/d")` is equivalent to `s("a", s(include = c("b", "c"), s("d", ...)))`

<code>as</code>	name for this field in the extracted data.frame
<code>children, ...</code>	Unnamed list of children spec. ... is merged into children. children is part of the canonical spec.
<code>groups</code>	Named list of specs to be processed in parallel. The return value is a named list of unnested data.frames. The results is the same as when each spec is unnested separately except that <code>dedupe</code> parameter of <code>unnest()</code> will work across groups and execution is faster because the nested list is traversed once regardless of the number of groups.
<code>include, exclude</code>	A list, a numeric vector or a character vector specifying components to include or exclude. A list can combine numeric indexes and character elements to extract.
<code>stack</code>	Whether to stack this node (TRUE) or to spread it (FALSE). When <code>stack</code> is a string an index column is created with that name.
<code>process</code>	Extra processing step for this element. Either NULL for no processing (the default), "as_is" to return the entire element in a list column, "paste" to paste elements together into a character column.
<code>default</code>	Default value to insert if the <code>include</code> specification hasn't matched.

## Value

`s()`: a canonical spec - a list consumed by C++ unnesting routines.

## Examples

```
s("a")
s("a//c2")
s("a/2/c2,cid")
```

---

unnest

*Unnest lists*


---

## Description

Unnest nested lists into a flat data.frames.

**Usage**

```
unnest(
  x,
  spec = NULL,
  dedupe = FALSE,
  stack_atomic = NULL,
  process_atomic = NULL,
  process_unnamed_lists = NULL,
  cross_join = TRUE
)
```

**Arguments**

<code>x</code>	a nested list to unnest
<code>spec</code>	spec to use for unnesting. See <a href="#">spec()</a> .
<code>dedupe</code>	whether to dedupe repeated elements. If <code>TRUE</code> , if a node is visited for a second time and is not explicitly declared in the spec the node is skipped. This is particularly useful with grouped specs.
<code>stack_atomic</code>	Whether atomic leaf vectors should be stacked or not. If <code>NULL</code> , the default, <code>data.frame</code> vectors are stacked, all others are spread.
<code>process_atomic</code>	Process spec for atomic leaf vectors. Either <code>NULL</code> for no processing (the default), <code>"as_is"</code> to return the entire element in a list column, <code>"paste"</code> to paste elements together into a character column.
<code>process_unnamed_lists</code>	How to process unnamed lists. Can be one of <code>"as_is"</code> - return a list column, <code>"exclude"</code> - drop these elements unless they are explicitly included in the spec, <code>"paste"</code> - return a character column, <code>"stack"</code> - automatically stack. If <code>NULL</code> (the default), do nothing - process them normally according to the specs.
<code>cross_join</code>	Specifies how the results from sibling nodes are joined (cbinded) together. The shorter <code>data.frames</code> (fewer rows) can be either recycled to the max number of rows across all joined components with <code>cross_join = FALSE</code> . Or, the results are cross joined (produce all combinations of rows across all components) with <code>cross_join = TRUE</code> . <code>cross_join = TRUE</code> is the default because of no data loss and it is more conducive for earlier error detection with incorrect specs

**Value**

A `data.frame`, `data.table` or a `tibble` as specified by the option `unnest.return.type`. Defaults to `data.frame`.

**Examples**

```
x <- list(a = list(b = list(x = 1, y = 1:2, z = 10),
                  c = list(x = 2, y = 100:102)))
xxx <- list(x, x, x)

## spreading
```

```

unnest(x, s("a"))
unnest(x, s("a"), stack_atomic = TRUE)
unnest(x, s("a/b"), stack_atomic = TRUE)
unnest(x, s("a/c"), stack_atomic = TRUE)
unnest(x, s("a"), stack_atomic = TRUE, cross_join = TRUE)
unnest(x, s("a//x"))
unnest(x, s("a//x,z"))
unnest(x, s("a/2/x,y"))

## stacking
unnest(x, s("a/", stack = TRUE))
unnest(x, s("a/", stack = TRUE, as = "A"))
unnest(x, s("a/", stack = TRUE, as = "A"), stack_atomic = TRUE)
unnest(x, s("a/", stack = "id"), stack_atomic = TRUE)
unnest(x, s("a/", stack = "id", as = ""), stack_atomic = TRUE)

unnest(xxx, s(stack = "id"))
unnest(xxx, s(stack = "id", stack_atomic = TRUE))
unnest(xxx, s(stack = "id", s("a/b/y/", stack = TRUE)))

## exclusion
unnest(x, s("a/b/", exclude = "x"))

## dedupe
unnest(x, s("a", s("b/y"), s("b")), stack_atomic = TRUE)
unnest(x, s("a", s("b/y"), s("b")), dedupe = TRUE, stack_atomic = TRUE)

## grouping
unnest(xxx, stack_atomic = TRUE,
      s(stack = TRUE,
        groups = list(first = s("a/b/x,y"),
                      second = s("a/b"))))

unnest(xxx, stack_atomic = TRUE, dedupe = TRUE,
      s(stack = TRUE,
        groups = list(first = s("a/b/x,y"),
                      second = s("a/b"))))

## processing as_is
str(unnest(xxx, s(stack = "id",
                s("a/b/y", process = "as_is"),
                s("a/c", process = "as_is"))))
str(unnest(xxx, s(stack = "id", s("a/b/", process = "as_is"))))
str(unnest(xxx, s(stack = "id", s("a/b", process = "as_is"))))

## processing paste
str(unnest(x, s("a/b/y", process = "paste")))
str(unnest(xxx, s(stack = TRUE, s("a/b/", process = "paste"))))
str(unnest(xxx, s(stack = TRUE, s("a/b", process = "paste"))))

## default
unnest(x, s("a/b/c/", s("b", default = 100)))
unnest(x, s("a/b/c/", stack = "ix", s("b", default = 100)))

```



# Index

s (spec), 2

spec, 2

spec(), 4

unnest, 3