# Package: shinyTree (via r-universe)

October 28, 2026

**Type** Package

**Title** 'jsTree' Bindings for 'Shiny'

**Version** 1.0.0

**Maintainer** Michael Bell <bell_michael_a@lilly.com>

**Description** Bindings to 'jsTree' <https://www.jstree.com>, a 'JavaScript' library for interactive trees, to enable rich, editable trees in 'Shiny' applications. Trees can be searched, checked, rearranged, styled and remembered across visits, loaded lazily for very large hierarchies, and built from data frames, 'data.tree' objects or ontologies.

**License** MIT + file LICENSE

**Copyright** See the file COPYRIGHTS.

**Depends** R (>= 3.3.0)

**Imports** shiny (>= 1.6.0), htmltools, jsonlite, promises, stats, tools, utils

**Suggests** testthat, shinytest2, covr, data.tree, mockery, withr, later, rmarkdown, xml2, future, shinyjs, tidyr

**URL** https://github.com/shinyTree/shinyTree

**BugReports** https://github.com/shinyTree/shinyTree/issues

**RoxygenNote** 7.3.3

**Encoding** UTF-8

**Language** en-GB

**NeedsCompilation** no

**Author** Trestle Technology, LLC [aut], Jeff Allen [aut], Institut de Radioprotection et de Sûreté Nucléaire [cph], Ivan Bozhanov [ctb, cph] (jsTree), Orange Hill Development [ctb, cph] (jsTree Bootstrap (proton) theme), Accademia di Belle Arti di Urbino [cph] (Titillium Web font), Mike Schaffer [ctb], Timm Danker [ctb], Michael Bell [cre], Sebastian Gatscha [ctb], Thorn Thaler [ctb]

**Repository** https://cran.r-universe.dev

**Date/Publication** 2026-10-28 10:30:02 UTC

**RemoteUrl** https://github.com/cran/shinyTree

**RemoteRef** HEAD

**RemoteSha** fdf34b9dfa461b7f3f2ae58b5b4b1ce736bf12a5

# Contents

dfToTree . . . 2
expandTree . . . 3
get_checked . . . 4
get_selected . . . 5
ontologyToTree . . . 6
renderEmptyTree . . . 8
renderTree . . . 8
renderTreeAsync . . . 12
set_node_attrs . . . 13
shinyTree . . . 14
treeToDf . . . 17
treeToJSON . . . 18
updateTree . . . 19

**Index** **21**

---

`dfToTree` *Convert a data.frame to a nested list for renderTree*

---

## Description

Each column of `hierarchy` is one level of the tree, in that order: a node for each value of the first column, under it a node for each value of the second column found with it, and so on. A missing value ends the branch there, which is how `treeToDf` writes a branch shorter than the deepest one, so a tree survives the round trip. A row whose first column is missing is left out.

## Usage

```
dfToTree(df, hierarchy = colnames(df))
```

## Arguments

| | |
|---|---|
| `df` | data.frame |
| `hierarchy` | ordered character vector of column names defining the hierarchy |

## Value

A named nested list, as `renderTree` takes; an empty list for a data frame with no rows.

## Author(s)

Jasper Schelfhout <jasper.schelfhout@openanalytics.eu>

## See Also

treeToDf, the 22-dfToTree example

## Examples

```
df <- data.frame(Titanic)
tree <- dfToTree(df, c("Sex", "Class", "Survived"))
str(tree$Male$`1st`)
```

---

expandTree *Expand or collapse every node of a tree*

---

## Description

Opens or closes every node of a shinyTree in the browser, for example from "Expand all" and "Collapse all" buttons. The tree's contents are not resent, so selection, checkboxes and search are kept, and input$tree reports each node's new open state as it would after a user click.

## Usage

```
expandTree(session, treeId)

collapseTree(session, treeId)
```

## Arguments

| | |
|---|---|
| session | The current session variable. |
| treeId | The identifier for the shinyTree object. |

## Details

To control which nodes start open instead, set stopened = TRUE on them when rendering the tree.

expandTree cannot be used on a tree rendered with renderTree(lazy = TRUE), since it would have to fetch the whole tree, and stops with an error. collapseTree works on any tree.

## Value

Called for its side effect; returns NULL invisibly.

## See Also

updateTree

## Examples

```
if (interactive()) {
  library(shiny)
  ui <- fluidPage(
    actionButton("expand", "Expand all"),
    actionButton("collapse", "Collapse all"),
    shinyTree("tree")
  )
  server <- function(input, output, session) {
    output$tree <- renderTree(list(a = list(b = list(c = ""))))
    observeEvent(input$expand, expandTree(session, "tree"))
    observeEvent(input$collapse, collapseTree(session, "tree"))
  }
  shinyApp(ui, server)
}
```

---

`get_checked` *Get the checked nodes from a tree*

---

## Description

Extract the nodes from the tree that are checked in a more convenient format. You can choose which format you prefer.

## Usage

```
get_checked(
  tree,
  format = c("names", "slices", "classid"),
  leaves_only = FALSE
)
```

## Arguments

| | |
|---|---|
| `tree` | The `input$tree` shinyTree you want to inspect. |
| `format` | In which format you want the output. Use `names` to get a simple list of the names (with attributes describing the node's ancestry), or `slices` to get a list of lists, each of which is a slice of the list used to get down to the checked node. `classid` gives each name with its `stclass`, `stid` and `id` (jsTree's id for the node) attributes instead of its ancestry. |
| `leaves_only` | If `TRUE`, return only checked nodes that have no children, leaving out checked parents. With `three_state = TRUE` a parent is checked whenever all of its children are, so this gives just the items that were ticked. |

## Value

A list with one element per checked node, descendants before their ancestors, in the form `format` gives. An empty list if nothing is checked or `tree` is `NULL`.

## See Also

get_selected

## Examples

```
tree <- list(
  Fruit = structure(list(
    Apple = structure(0, stchecked = TRUE),
    Pear = 0
  ), stopened = TRUE)
)
unlist(get_checked(tree))
```

---

# get_selected *Get the selected nodes from a tree*

---

## Description

Extract the nodes from the tree that are selected in a more convenient format. You can choose which format you prefer.

## Usage

```
get_selected(
  tree,
  format = c("names", "slices", "classid"),
  leaves_only = FALSE
)
```

## Arguments

| | |
|---|---|
| tree | The input$tree shinyTree you want to inspect. |
| format | In which format you want the output. Use names to get a simple list of the names (with attributes describing the node's ancestry), or slices to get a list of lists, each of which is a slice of the list used to get down to the selected node. classid gives each name with its stclass, stid and id (jsTree's id for the node) attributes instead of its ancestry. |
| leaves_only | If TRUE, return only selected nodes that have no children, leaving out selected parents. This works however deep each branch goes, unlike filtering slices by depth. |

## Value

A list with one element per selected node, descendants before their ancestors, in the form format gives. An empty list if nothing is selected or tree is NULL.

## See Also

get_checked

## Examples

```
tree <- list(
  World = structure(list(
    Europe = structure(list(
      Scandinavia = structure(list(
        Denmark = structure(0, stselected = TRUE),
        Sweden = structure(0, stselected = TRUE)
      ), stselected = TRUE),
      France = structure(0, stselected = TRUE)
    ), stselected = TRUE)
  ), stselected = TRUE)
)
unlist(get_selected(tree))
unlist(get_selected(tree, leaves_only = TRUE))
```

---

# ontologyToTree *Show an ontology as a tree*

---

## Description

Reads an ontology from an OBO (`.obo`) or OWL (`.owl`, RDF/XML) file and returns a children function for `renderTree(lazy = TRUE)`, so even a large ontology such as the Gene Ontology is shown a level at a time. Reading OWL needs the **xml2** package.

## Usage

```
ontologyToTree(file, format = NULL, obsolete = "obsolete")
```

## Arguments

| | |
|---|---|
| `file` | Path to an `.obo` or `.owl` file. |
| `format` | `"obo"` or `"owl"`; by default guessed from the file's extension: `.obo` is OBO, and `.owl`, `.rdf` and `.xml` are OWL. |
| `obsolete` | Label of the top-level node holding the obsolete terms, or `NULL` to leave them out. |

## Details

The hierarchy is formed by `is_a` (in OWL, `rdfs:subClassOf` between named classes). Each node is labelled with the term's name and carries its identifier as `stid` and its definition, if any, as `sttooltip`, so it is shown on hover.

**Several parents** A term with more than one parent is shown under each of them, with the same `stid` in every place, and search finds it in each.

**Obsolete terms** Terms marked obsolete (`is_obsolete: true`, or `owl:deprecated`) have no parent, so they are gathered under one extra top-level node, `obsolete`, and shown greyed out (`stdisabled`). With `obsolete = NULL` they are left out, and a live term whose only parents are obsolete is shown at the top level.

**Terms left out** OWL classes with no label, and OBO stanzas with no name and no parent, are placeholders – typically left behind when terms are merged or replaced – and are not shown.

**Parents outside the file** Only terms in the file are shown. A term whose parents are all outside it – in an ontology it imports, say – appears at the top level.

**OBO and OWL** The OWL release of an ontology usually includes every class it imports from other ontologies, which its OBO release leaves out, so the two can have different top levels. The OBO file usually gives the tidier tree.

**Cycles** A term caught in a cycle of `is_a` – A is_a B, B is_a A – with no way up to a root is shown at the top level, so nothing in the file is lost.

**Names** Siblings can share a name; they are told apart by `stid`, so the function takes `node` as well as `path`. Names are shown as they are in the file, as HTML (see `renderTree`).

**Size** The Gene Ontology's 48,000 terms are read in a few seconds, and each branch is fetched only when opened. A branch with thousands of children, such as its obsolete terms, takes a few seconds to open.

## Value

A list for `renderTree(lazy = TRUE)`: `children`, the function giving a node's children, and `search`, which finds terms whose name or identifier contains the search text and returns the path to each place a matching term appears (a term with several parents appears in several). It also has an attribute `"ontology"`: a data frame of the terms, with columns `id`, `name`, `definition` and `obsolete`, and a list column `parents`.

## See Also

`renderTree`, whose *Lazy loading* section describes the children function

## Examples

```
# The Anatomical Entity Ontology, which comes with the 28-ontology example
aeo <- ontologyToTree(system.file("examples/28-ontology/aeo.obo", package = "shinyTree"))
names(aeo$children(character(0)))
head(attr(aeo, "ontology")[, c("id", "name")])

if (interactive()) {
  library(shiny)
  ui <- fluidPage(shinyTree("tree", search = TRUE))
  server <- function(input, output, session) {
    output$tree <- renderTree(lazy = TRUE, aeo)
  }
  shinyApp(ui, server)
}

# The 28-ontology example browses this ontology, or one you upload, such as
# the Gene Ontology (go-basic.obo from https://geneontology.org):
```

```
if (interactive()) {
  shiny::runApp(system.file("examples/28-ontology", package = "shinyTree"))
}
```

---

`renderEmptyTree` *Render an empty ShinyTree*

---

## Description

Renders a tree with no defined nodes.

## Usage

```
renderEmptyTree()
```

## Value

A render function, to assign to `output$<id>`.

## See Also

`shinyTree`

---

`renderTree` *Render a ShinyTree*

---

## Description

Fills a `shinyTree` with the tree an expression returns, as a nested list (see *Node attributes*) or as JSON in jsTree's format, such as `treeToJSON` gives.

## Usage

```
renderTree(expr, env = parent.frame(), quoted = FALSE, lazy = FALSE)
```

## Arguments

| | |
|---|---|
| `expr` | An expression that returns the tree: a named nested list, or a JSON string. With `lazy = TRUE`, a function instead. |
| `env` | The environment in which `expr` should be evaluated. |
| `quoted` | Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable. |
| `lazy` | If `TRUE`, the tree's nodes are fetched as they are opened rather than all at once, for trees too large to send whole. `expr` then returns a function giving a node's children; see *Lazy loading*. |

## Value

A render function, to assign to `output$<id>` for a `shinyTree` with that id.

## Note

A tree's contents are pushed to the client rather than returned as an output value, so Shiny has nothing cached to replay if the element is removed from the page and later added back – inside a `modalDialog` that is dismissed and reopened, for example, or a conditional panel. The client reports whenever it binds an empty tree and the contents are resent, so such a tree repopulates without the app having to re-trigger the render itself.

## Node attributes

Each node is an element of the list, named by its label. Attributes set on it with `structure` or `attr()` control how it is shown:

`stopened` `TRUE` to show the node expanded.

`stselected` `TRUE` to select it.

`stchecked` `TRUE` to tick its checkbox when `tie_selection = FALSE`. With the default `tie_selection = TRUE` a ticked box is a selection, so use `stselected` instead.

`stdisabled` `TRUE` to disable it.

`sticon` an icon: Font Awesome classes such as `"fa fa-file"` or `"fa-solid fa-folder"`, or the URL or path of an image. A name on its own, `"file"` or `"fa-file"`, is expanded to `"fa fa-file"`, and `"glyphicon-file"` to `"glyphicon glyphicon-file"`. Glyphicons need Bootstrap 3, the default for `fluidPage()`; any other icon set works if the app loads its stylesheet.

`sttype` the node's type, for the `types` argument of `shinyTree`.

`sttooltip` text shown when the pointer rests on the node's label. It is shown as plain text, not HTML.

`stcssclass` one or more CSS classes for the node, as a character vector or a space-separated string, so the app's own CSS can style it. They go on the node's `<li>`, so target its label with `li.myclass > a`. Unlike `sttype`, any number can be given.

Any other attribute is passed through as data and comes back on the node in `input$tree`, apart from those R itself gives a node, such as a data frame's `class` and `row.names`. `get_selected` and `get_checked` keep those whose names start with `st`, so that is the convention to follow. That includes `stclass`, which despite its name is only data – it is what `get_selected(format = "classid")` reports – and is not applied as a CSS class; use `stcssclass` for that.

Every node needs a name, and names are shown as HTML, so a name read from a file or a database can contain markup. Siblings may share a name.

## Lazy loading

With `lazy = TRUE`, `expr` returns a function that is given a node's `path` – the names from the root down to it, `character(0)` for the top level – and returns that node's children as a named list, in the same form `renderTree()` takes. Only the top level is sent when the tree is shown; each node's children are fetched when the user opens it.

```
output$tree <- renderTree(lazy = TRUE, {
  root <- input$folder          # reactive: a change reloads the whole tree
  function(path) {
    dir <- do.call(file.path, as.list(c(root, path)))
    subdirs <- list.dirs(dir, full.names = FALSE, recursive = FALSE)
    # A symbolic link can point back up the tree; show it as a leaf.
    links <- nzchar(Sys.readlink(file.path(dir, subdirs)))
    files <- setdiff(list.files(dir), subdirs[!links])
   c(setNames(lapply(subdirs[!links], function(d) structure("", stchildren = TRUE)),
              subdirs[!links]),
      setNames(lapply(files, function(f) ""), files))
  }
})
```

`stchildren = TRUE` marks a child that has children of its own, so it shows as closed and is fetched when opened. A child given as a non-empty list is sent with its children straight away instead. Any value other than `TRUE` or `FALSE` is treated as `FALSE`, with a warning.

`function(path, node)` The function may take an argument named `node`: the `st*` attributes of the node being opened, as a list. Siblings with the same name have the same path, so give each an id, `structure("", stchildren = TRUE, stid = 42)`, and use `node$stid` to tell them apart. Only an argument called `node` receives it. `stpath` is used by shinyTree for the path a node was fetched under, so do not set it yourself.

**Node attributes** All the attributes in *Node attributes* above work on the children, including `stselected` and `stopened`; an opened child's children are fetched at once.

**Reactivity** `expr` is evaluated as usual, so anything reactive it reads rebuilds the tree when it changes, which starts it again from the top level. The children function can read reactive values too, and gets their current value, but reading one there does not rebuild the tree.

**Errors** An error in the children function is shown in the R console, the node it was opening stays closed so opening it again retries, and `input$<outputId>_lazy_error` is set to a list with the node's `path` and the `message`. With `options(shiny.sanitize.errors = TRUE)` the message is replaced by a generic one, as for other outputs, unless it is a `safeError`.

**Search** With `shinyTree(search = TRUE)` the whole tree is searched, not just the nodes fetched so far: the server finds the matches, and the branches leading to them are fetched and opened. By default this walks the whole tree with the children function, matching names case-insensitively, and gives up with an error after 10,000 calls or 100,000 nodes or 5 seconds. A branch whose children function fails is skipped. For anything larger, or slow, return `list(children = <function>, search = <function>)` instead, where `search` is given the search text and returns the paths of the matching nodes as a list of character vectors, e.g. from a database. At most 1,000 matches are shown.

`input$tree` holds the nodes fetched so far. Nodes can only be selected or ticked once they are fetched, so `get_selected()` and `get_checked()` are complete – except that ticking a branch that has never been opened reports the branch, not the children it has not fetched, so `leaves_only = TRUE` cannot list them.

**State** With `shinyTree(state = TRUE)`, the branches that were open, and the parents of the nodes that were selected or ticked, are fetched and reopened after a reload. Of several siblings with the same name, the same one is reopened, by its position among them.

**Renaming and moving** A node keeps the path it was fetched under, so after the user renames or moves it with the context menu or drag and drop, opening it still asks for its children by its original path.

**Speed** One R process answers every user of the app, so a children function that takes a second makes everyone wait a second. Keep it fast: query only the one level asked for, and cache expensive results.

**Trust** The server only asks the children function for branches it has sent to that browser – never for a leaf, and never for a path it made up, such as `c("..", "..")` – and `node` is always the data the server sent with that branch, never what the browser says it is. It cannot know what a name means to the app, though: if the app turns names into file paths or database keys, it is still up to the app not to offer a branch, such as a symbolic link, that leads somewhere it should not.

**Not available** `bindCache()`; `updateTree()`, which stops with an error – change what `expr` returns to rebuild the tree; and `expandTree()`, which stops with an error since it would have to fetch the whole tree.

Several branches that need fetching at once – on a search or a state restore – are fetched in a single request. The `27-lazy` example shows a million-node tree with a `search` function, and a folder browser.

## Caching

`renderTree` can be wrapped in `bindCache`, in which case `expr` is evaluated only for a cache key that has not been seen before and the cached tree is sent otherwise. It works with `bindEvent` as well.

## See Also

`shinyTree`

## Examples

```
if (interactive()) {
  library(shiny)
  ui <- fluidPage(shinyTree("tree"), verbatimTextOutput("picked"))
  server <- function(input, output, session) {
    output$tree <- renderTree(list(
      Fruit = structure(list(Apple = "", Pear = structure("", stselected = TRUE)),
                        stopened = TRUE, sticon = "fa fa-folder"),
      Vegetables = list(Leek = "")
    ))
    output$picked <- renderPrint(get_selected(input$tree))
  }
  shinyApp(ui, server)
}
```

renderTreeAsync *Render a ShinyTree asynchronously*

## Description

Like renderTree, but expr may also return a promise (see the **promises** package) of the tree, for a tree that is slow to build: with **future**, it is built in another process and the app stays responsive in the meantime. The tree is sent once the promise resolves; if it is rejected, the error is shown in place of the tree. An expression that returns a tree straight away works as it does in renderTree.

## Usage

```
renderTreeAsync(expr, env = parent.frame(), quoted = FALSE)
```

## Arguments

| | |
|---|---|
| expr | An expression that returns a nested list or JSON, as for renderTree, or a promise of one. |
| env | The environment in which expr should be evaluated. |
| quoted | Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable. |

## Details

Lazy loading (renderTree(lazy = TRUE)) is not available here.

## Value

A render function, to assign to output$<id> for a shinyTree with that id.

## See Also

renderTree, shinyTree

## Examples

```
if (interactive() && requireNamespace("future", quietly = TRUE)) {
  library(shiny)
  library(promises)
  library(future)
  plan(multisession)

  ui <- fluidPage(shinyTree("tree"))
  server <- function(input, output, session) {
    output$tree <- renderTreeAsync({
      future({
        Sys.sleep(2)  # a tree that is slow to build
        list(root = list(leaf = ""))
```

```
      })
    })
  }
  shinyApp(ui, server)
}

# See also the 16-async example:
if (interactive()) {
  shiny::runApp(system.file("examples/16-async", package = "shinyTree"))
}
```

---

`set_node_attrs` *Set an attribute on every inner node and leaf of a tree*

---

## Description

Traverse through tree/list to set node attributes, e.g. change icons. Useful for directory structure icons where inner nodes are directories, leaves are files.

## Usage

```
set_node_attrs(tree, attr_name, inner_val, leaf_val)
```

## Arguments

| | |
|---|---|
| `tree` | named nested list |
| `attr_name` | name of attribute to set |
| `inner_val` | value of attribute for inner tree nodes |
| `leaf_val` | value of attribute for leaf nodes |

## Value

named nested list

## Examples

```
tree <- dfToTree(data.frame(Titanic),  c("Sex", "Survived"))
str(set_node_attrs(tree, attr_name = "sttype", inner_val = "directory", leaf_val = "file"))
```

shinyTree *Create a Shiny Tree*

## Description

This creates a spot in your Shiny UI for a shinyTree which can then be filled in using `renderTree`.

## Usage

```
shinyTree(
  outputId,
  checkbox = FALSE,
  search = FALSE,
  searchtime = 250,
  searchplaceholder = "",
  dragAndDrop = FALSE,
  types = NULL,
  theme = "default",
  themeIcons = TRUE,
  themeDots = TRUE,
  sort = FALSE,
  unique = FALSE,
  wholerow = FALSE,
  stripes = FALSE,
  multiple = TRUE,
  animation = 200,
  contextmenu = FALSE,
  three_state = TRUE,
  whole_node = TRUE,
  tie_selection = TRUE,
  state = FALSE,
  state_priority = c("saved", "code")
)
```

## Arguments

| | |
|---|---|
| `outputId` | The ID associated with this element |
| `checkbox` | If `TRUE`, will enable checkboxes next to each node to make the selection of multiple nodes in the tree easier. |
| `search` | If `TRUE`, will enable search functionality in the tree by adding a search box above the produced tree. Alternatively, you can set the parameter to the ID of the text input you wish to use as the search field. |
| `searchtime` | How long to wait, in milliseconds, after the user stops typing before searching. Default is 250ms. |
| `searchplaceholder` | Add a placeholder value to the search box. Only used when `search = TRUE`, since a search field given by its id has its own. |

| | |
|---|---|
| dragAndDrop | If TRUE, will allow the user to rearrange the nodes in the tree. |
| types | A JavaScript object literal, as a string, giving jsTree's types settings: each type's icon, valid children and so on. Nodes are given a type with the sttype attribute. It is written into the page as JavaScript, not parsed as JSON, so single quotes work; see the 12-types example and https://www.jstree.com/plugins/. |
| theme | jsTree theme, one of default, default-dark, or proton. |
| themeIcons | If TRUE, will show theme icons for each item. |
| themeDots | If TRUE, will include level dots. |
| sort | If TRUE, will sort the nodes in alphabetical/numerical order. |
| unique | If TRUE, will ensure that no node name exists more than once. |
| wholerow | If TRUE, will highlight the whole selected row. |
| stripes | If TRUE, the tree background is striped. |
| multiple | If TRUE, multiple nodes can be selected. |
| animation | The open / close animation duration in milliseconds. Set this to FALSE to disable the animation (default is 200). |
| contextmenu | If TRUE, will enable a contextmenu to create/rename/delete/cut/copy/paste nodes. jsTree cannot show checkboxes and the contextmenu together, so checkbox is then turned off, with a warning. |
| three_state | If TRUE, checking a node cascades to its children, and a parent is shown checked only while all of its children are (and as undetermined while some are). |
| whole_node | If TRUE, clicking anywhere on a node toggles its checkbox, as if the box itself were clicked. If FALSE, only the box does; clicking the label behaves as it would without checkboxes. |
| tie_selection | If TRUE, checked and selected are the same thing: ticking a box selects the node and selecting a node ticks its box, so get_checked and get_selected report the same nodes. If FALSE, checking is kept separate from the selection, so a node can be checked without being selected. With whole_node = TRUE as well, every click toggles a checkbox and nothing can be selected by clicking. |
| state | If TRUE, the browser remembers which nodes were open, selected and checked, and restores them when the page is reloaded. A string does the same and is used as the key the state is stored under, instead of the default "shinyTree-<outputId>". FALSE (the default) turns it off. See *Remembering state* below. |
| state_priority | Only used with state. When the remembered state disagrees with the stopened, stselected or stchecked attributes the tree was rendered with: "saved" (the default) restores what the user had; "code" keeps the app's attributes and restores the rest around them. |

## Details

A shinyTree is an output *and* an input element at the same time. While you can fill it via renderTree you can access its content via input$tree (for example after the user rearranged some nodes). By default, input$tree will return a list similar to the one you use to fill the tree. This behaviour is controlled by getOption("shinyTree.defaultParser"). It defaults to "list", but can be set to "tree", in which case a data.tree is returned (see *Options*).

Any valid Shiny id can be used, including one containing . or -, as Shiny modules produce.

## Value

A tag list holding the tree and the scripts and stylesheets it needs, for the app's UI.

## Checked and selected nodes

Without checkboxes, clicking a node selects it and `get_selected` reports the selection. With `checkbox = TRUE`, `tie_selection` decides whether a checked node counts as selected; see its description above. The `04-selected` example shows `get_selected` and `get_checked` side by side and can switch between every combination of `tie_selection`, `whole_node` and `three_state`.

## Remembering state

With `state` set, the browser remembers how the user left the tree and puts it back that way when the page is reloaded:

```
shinyTree("tree", checkbox = TRUE, state = TRUE)
```

**What is remembered** Which nodes are open, which are selected and, with `tie_selection = FALSE`, which are checked. It is saved each time one of these changes.

**When it is restored** Once, the first time the tree is filled after the page loads. A tree sent later by `updateTree` or by the render function running again is shown exactly as the server sent it, since it may be a different tree.

**How nodes are matched** By their path of names from the root, such as Europe / France / Paris. A remembered node is restored only if a node with that path still exists, so a node that has been renamed, moved or removed is simply not restored, and a remembered state can never be applied to an unrelated node. Where siblings share a name, each of them matches.

**Which wins** When the remembered state disagrees with the `stopened`, `stselected` or `stchecked` attributes the tree was rendered with, `state_priority = "saved"` (the default) restores what the user had, and `state_priority = "code"` keeps the app's attributes and restores the rest of the remembered state around them. With `"code"` a node the app opens or selects is always open or selected after a reload; with `"saved"` the user can close or deselect it and that is remembered.

**What the server sees** The restored state is reported in `input$tree` like any change the user makes, in a single update.

The state is kept in the browser's `localStorage`, so it is per browser and per user, survives closing the browser, and is never sent to the server. It is stored under `"shinyTree-<outputId>"` unless `state` is a string, which is used as the key instead. Pages from the same server share `localStorage`, so two apps there whose trees have the same id should pass different keys. To forget a remembered state, use a new key, or run `localStorage.removeItem("<key>")` in the browser.

## Options

`shinyTree.defaultParser` What `input$tree` holds: `"list"` (the default), a nested list like the one passed to `renderTree`, or `"tree"`, a `data.tree` Node, which needs the **data.tree** package. A data.tree cannot hold siblings with the same name, so the `"tree"` parser warns if the tree has any. It can be set before or after the package is loaded.

`shinyTree.refresh` `TRUE` to report `refresh.jstree` events. Default `FALSE`.

`shinyTree.setState` `TRUE` to report `set_state.jstree` events. Default `FALSE`.

The last two must be set before the UI is built, as they are written into the page. They only matter for refreshes started in the browser, e.g. by the app's own JavaScript: a tree sent by `renderTree` or `updateTree` is reported once it has loaded either way, and the refresh that loads it is not reported again – that caused a feedback loop in apps that call `updateTree` from an observer watching `input$tree`.

## See Also

`renderTree`

## Examples

```
if (interactive()) {
  library(shiny)

  # The tree comes back the way the user left it when the page is reloaded.
  # Europe starts open and Paris selected; with state_priority = "code" they
  # stay that way after every reload, and anything else the user opened or
  # selected is restored as well.
  ui <- fluidPage(
    shinyTree("tree", state = TRUE, state_priority = "code")
  )
  server <- function(input, output, session) {
    output$tree <- renderTree(list(
      Europe = structure(list(
        France = list(Paris = structure("", stselected = TRUE), Lyon = ""),
        Spain = list(Madrid = "")
      ), stopened = TRUE),
      Asia = list(Japan = list(Tokyo = ""))
    ))
  }
  shinyApp(ui, server)
}

# See also the 26-state example:
if (interactive()) {
  shiny::runApp(system.file("examples/26-state", package = "shinyTree"))
}
```

---

# treeToDf *Convert tree into data.frame*

---

## Description

The reverse of `dfToTree`: one row per leaf, with the names on the path down to it in the columns. A path shorter than the deepest one leaves the remaining columns `NA`.

## Usage

```
treeToDf(tree, hierarchy = NULL)
```

## Arguments

| | |
|---|---|
| tree | named nested list |
| hierarchy | ordered character vector with a column name for each level of the tree. If it is NULL or its length differs from the depth of the tree, the columns are named 1, 2, ... instead. |

## Value

data.frame

## Author(s)

Michael Bell

## See Also

dfToTree

## Examples

```
df <- data.frame(Titanic)
tree <- dfToTree(df, c("Sex", "Class", "Survived"))
newDf <- treeToDf(tree, c("Sex", "Class", "Survived"))
head(newDf)
```

---

# treeToJSON *Converts a data.tree to a JSON format*

---

## Description

Walk through a data.tree and constructs a JSON string, which can be rendered by shinyTree.

## Usage

```
treeToJSON(
  tree,
  keepRoot = FALSE,
  topLevelSlots = c("default", "all"),
  createNewId = TRUE,
  pretty = FALSE
)
```

### Arguments

| | |
|---|---|
| tree | the data.tree which should be parsed |
| keepRoot | logical. If FALSE (default) the root node from the tree is pruned |
| topLevelSlots | determines which slots should be moved to the top level of the node. If default or NULL slots used in the jsTree JSON are kept on the top level, while any other atomic / list slots from the tree are stored in an own slot called 'data'. If all, all slots are stored on the top level. Alternatively, it can be an explicit vector of slot names which should be kept. In the latter case it is the user's responsibility to ensure that jsTree slots stay on the top level. |
| createNewId | logical. If TRUE a new id will be generated. Any old 'id' will be stored in 'id.orig' and a warning will be issued. If FALSE, any existing id will be re-used. |
| pretty | logical. If TRUE the resulting JSON is prettified |

### Details

The JSON string generated follows the jsTree specifications. In particular it encodes children nodes via the 'children' slot.

All atomic or list slots of a node in the tree are stored in a data slot in the resulting JSON.

If the user wants to store some slots not in the data slot but on the top level of the node, parameter topLevelSlots can be used. This is useful for additional parameters such as 'icon', 'li_attr' or 'a_attr', which jsTree expects to be on the top level of the node.

An example of how to make use of this functionality can be found in the 19-data.tree example.

### Value

a JSON string representing the data.tree

### Note

updateTree and renderTree need an unevaluated JSON string. Hence, this function returns a string rather than the JSON object itself.

### Author(s)

Thorn Thaler, <thorn.thaler@thothal.at>

---

## updateTree *Update the tree with new data*

---

### Description

Replaces the contents of a shinyTree with a new tree, for example from an observer. Like the tree renderTree sends, the new one is reported in input$tree once it has loaded – unless it is the tree input$tree already holds, so an observer of input$tree that sends the tree back does not loop.

## Usage

```
updateTree(session, treeId, data = NULL)
```

## Arguments

| | |
|---|---|
| session | The current session variable. |
| treeId | The identifier for the shinyTree object |
| data | The new tree: a named nested list, as for renderTree, or JSON in jsTree's format, as from treeToJSON. NULL leaves the tree as it is. |

## Details

A tree rendered with `renderTree(lazy = TRUE)` cannot be updated this way, and `updateTree` stops with an error: change what its render expression returns instead. If the tree has not been shown yet – it is in a tab that has not been opened – the update is applied when it is.

## Value

Called for its side effect; returns NULL invisibly.

## See Also

renderTree, expandTree

## Examples

```
if (interactive()) {
  library(shiny)
  ui <- fluidPage(actionButton("reset", "Reset"), shinyTree("tree"))
  server <- function(input, output, session) {
    output$tree <- renderTree(list(a = list(b = ""), c = ""))
    observeEvent(input$reset, {
      updateTree(session, "tree", list(root = list(leaf = "")))
    })
  }
  shinyApp(ui, server)
}
```

# Index


bindCache, *11*
bindEvent, *11*

collapseTree *(*expandTree*)*, 3

data.tree, *15, 16*, *18*
dfToTree, 2, *17, 18*

expandTree, 3, *20*

get_checked, 4, *6*, *9*, *15*
get_selected, 5, *5*, *9*, *15, 16*

modalDialog, *9*

ontologyToTree, 6

renderEmptyTree, 8
renderTree, *2*, *6, 7*, 8, *12*, *14–17*, *19, 20*
renderTreeAsync, 12

safeError, *10*
set_node_attrs, 13
shinyTree, *3*, *8, 9*, *11, 12*, 14, *19*
structure, *9*

treeToDf, *2, 3*, 17
treeToJSON, *8*, 18, *20*

updateTree, *3*, *16, 17*, 19, *19*