Package: reactives (via r-universe)

October 15, 2026

Title Keyed Collections of 'shiny' Reactives

Version 0.1.0

Description Holds 'shiny' reactive values and reactive expressions under keys that can be added, removed and reordered, with dependencies tracked per key. The 'shiny' package offers reactiveValues() for keyed stored values, but it can neither remove nor reorder keys, and it has no keyed collection of computed values.

License GPL (>= 3)

Encoding UTF-8

Language en-US

RoxygenNote 8.0.0

Imports shiny (>= 1.14.0), stats, utils

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr, rmarkdown

Config/testthat/edition 3

URL https://nbenn.github.io/reactives/, https://github.com/nbenn/reactives

BugReports https://github.com/nbenn/reactives/issues

NeedsCompilation no

Author Nicolas Bennett [aut, cre, cph]

Maintainer Nicolas Bennett <nicolas@cynkra.com>

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

Date/Publication 2026-10-15 19:17:42 UTC

RemoteUrl https://github.com/cran/reactives

RemoteRef HEAD

RemoteSha 9c075ba66716bf680b85494cf629fbe8f17b32d0

Contents

reactives 2

Index **7**

`reactives` *Keyed collections of reactives*

Description

A `reactives` object is a keyed collection of shiny reactives, such as `shiny::reactiveVal()` and `shiny::reactive()` objects. Slots can be added, removed and reordered, and each slot is tracked as its own reactive dependency. The `reactive_vals()` constructor creates a collection whose slots are `shiny::reactiveVal()` objects holding the given values, and which only accepts `shiny::reactiveVal()` slots, so code writing through its slots can rely on them being writable. Its counterpart, `reactive_exprs()`, creates a collection that only accepts reactive expressions, the reactives of class `reactiveExpr` that `shiny::reactive()`, `shiny::eventReactive()`, `shiny::debounce()` and the like return, so that none of its slots can be written. To test whether an object is a collection of any kind, use `is_reactives()`.

Usage

```
reactives(...)

reactive_vals(...)

reactive_exprs(...)

## S3 method for class 'reactives'
reorder(x, order, ...)

append_reactive(x, value)

append_value(x, value)

is_reactives(x)

as_values(x)

has_key(x, key)

snapshot_reactives(x)
```

Arguments

`...`	For `reactives()`, `reactive_vals()` and `reactive_exprs()`, the slots to hold, named or unnamed: reactives for `reactives()` and reactive expressions for

	reactive_exprs(), with NULL entries dropped, and any values, including NULL, for reactive_vals(). Ignored by reorder().
x	A reactives object, or for is_reactives(), any object.
order	The new order, listing every slot exactly once: by position, or by name when all slots are named.
value	For append_reactive(), the reactive to bind to the new slot, and for append_value(), the value for the new slot's shiny::reactiveVal() to hold.
key	The names of the slots to test for, none of them empty or missing.

Details

A collection has two levels, as a list does: the slots, and the values their reactives return. Reading with $ or [[, as in x$a, x[["a"]] or x[[1]], returns a slot's value, as it does for a shiny::reactiveValues() object, while reading with [, as in x["a"] or x[1], returns the slot's reactive. As on a list, a name with no slot reads as NULL, while a position past the last slot is an error. A slot whose shiny::reactiveVal() stores NULL reads as NULL with $ and [[too, but not with [, which tells it apart from a missing slot. To test for slots by name, use has_key().

Each read takes a single name or position. Take the reactives of several slots from as.list(x)[i] instead, or get the value of every slot at once with as_values(), as shiny::reactiveValuesToList() does for a shiny::reactiveValues() object. Going through the slots' reactives works with lapply() and vapply(), which call as.list(), while Map() reads the slots by position with [[and so goes through their values. A for loop, do.call() and purrr's map() functions don't dispatch on the class, though, so they see the collection's internal fields instead of its slots. With those, go through as.list(x) instead, as in for (slot in as.list(x)).

Assigning a reactive with [<-, as in x["a"] <- r, binds it to the slot, adding the slot or replacing its reactive, and assigning NULL with [<- removes the slot, as for a list. Assigning anything else with [<- is an error. Assigning with $<- or [[<- writes a value instead, as for a shiny::reactiveValues() object: through the slot's shiny::reactiveVal(), which stays bound, or where there is no slot, by binding a new shiny::reactiveVal() that holds the value. Any value is stored as it is, a reactive included, and unlike on a list, assigning NULL with $<- or [[<- stores NULL rather than removing the slot. A slot holding a reactive expression, such as a shiny::reactive(), can't be written, so that a stray value can't overwrite it; replacing its reactive takes [<-. A reactive_exprs collection only holds such slots, and assigning a value to it with $<- or [[<- is an error even where there is no slot, so that a stray value can't add one either. As with reading, each assignment takes a single name or position.

Slots can be unnamed. To append one, bind a reactive r with append_reactive(x, r), or write a value with append_value(x, value), which binds a new shiny::reactiveVal() holding it. The checks are those of binding and writing, so append_value() is an error for a reactive_exprs collection.

Renaming with names<- is an error, as it is for a shiny::reactiveValues() object. To rename a slot, bind its reactive under the new name, remove the old slot and restore the order with reorder().

Value

A reactives object, or for reactive_vals() and reactive_exprs() an object of that class, which is also a reactives object. A snapshot has the class of x. The reorder() method returns x,

invisibly, and so do `append_reactive()` and `append_value()`. The `as_values()` function returns a list of the slots' values, in slot order and named as by `as.list()`, `has_key()` returns a logical vector with one element per name in `key`, and `is_reactives()` returns `TRUE` or `FALSE`.

Sharing

A collection is shared, like `shiny::reactiveValues()`: after `y <- x`, both names refer to the same collection, and a change made through `[<-`, `$<-`, `[[<-`, `reorder()`, `append_reactive()` or `append_value()` is seen by everything holding it. That is what lets one part of an app change a collection while another reacts to it.

Lifetime

A collection belongs to the session or module it was created in and is destroyed along with it, as a `shiny::reactiveVal()` is. Reading or changing it from another module doesn't tie it to that module, so destroying the module leaves the collection intact. A reactive bound to a slot, however, still belongs to the module it was created in, while the `shiny::reactiveVal()` that `$<-`, `[[<-` or `append_value()` creates for a new slot belongs to the collection, whichever module writes the value.

To keep reading a collection after its `shiny::testServer()` session has closed, a test can take a snapshot with `snapshot_reactives()` while the session is still open. A snapshot belongs to no session. For each `shiny::reactiveVal()` slot it holds a new `shiny::reactiveVal()` with the slot's current value, and for each other slot a `shiny::reactive()` returning its current value, so writing through a slot of the snapshot or of the original leaves the other unchanged. A slot that fails when computed fails the same way when its copy is called, rather than failing the snapshot. Names, order and class carry over.

Dependencies

Reading a slot's reactive by name, as in `x["a"]`, makes the caller depend on that slot alone. The caller re-runs when the slot is bound, replaced or removed, but not when other slots change. This also holds for a slot that does not exist yet, so a reader re-runs once the slot is added. Reading the slot's value, as in `x$a` or `x[["a"]]`, depends on the slot and on its value, as calling its reactive does, so the caller also re-runs when the value changes. Testing for slots with `has_key()` has the same dependencies as reading their reactives by name, while testing with `"a" %in% names(x)` depends on what `names()` depends on, so the caller re-runs whenever a slot is added, removed or moved.

The `length()` method depends on the number of slots alone, and `names()` on which slots exist and their order. Reading a slot by position, as in `x[1]` or `x[[1]]`, depends on what `names()` depends on and on the slot it finds, and `x[[1]]` also on its value, so the caller re-runs whenever a slot is added, removed or moved, even if the slot at its position stays the same. Reading past the last slot fails, and the caller re-runs on the same changes. The `as.list()` method depends on what `names()` depends on and on every slot, and `as_values()` also on every slot's value. Reordering re-runs readers of `names()`, `as.list()`, `as_values()` and of slots by position, but not readers of `length()` or of slots by name.

To depend on the slot at a position alone, read it through a `shiny::reactiveVal()`. After `first <- reactiveVal()` and `observe(first(if (length(x)) x[1]))`, a reader of `first()` re-runs only when position 1 comes to hold a different reactive or none, since writing a `shiny::reactiveVal()`

the value it already holds invalidates nothing. The check on `length()` keeps the observer from failing on an empty collection, which in an app would end the session.

Binding, removing and writing make the caller depend on nothing, and so does appending with `append_reactive()` or `append_value()`. Writing a value through a slot's `shiny::reactiveVal()` leaves the slot bound to it, so the write re-runs only the readers of the slot's value, and only if the value changes. Writing a value where there is no slot binds a new one and re-runs readers as any binding does.

Assigning one past the last slot, as in `x[length(x) + 1] <- r` or `x[[length(x) + 1]] <- value`, appends as well, but the call to `length()` makes the caller depend on the number of slots, which the append then changes. An observer that appends this way re-runs after each append and appends again, without end.

Printing, `format()` and `str()` make the caller depend on nothing, and they never call a slot, so they run no computed slot. Taking a snapshot also makes the caller depend on nothing, though it runs every computed slot. Outside a reactive consumer, as at the console, `names()` and `length()` work as they would inside `shiny::isolate()` rather than fail.

Reactlog

In reactlog, the dependency on slot a shows as `reactives$a`, or as `reactive_vals$a` or `reactive_exprs$a` in a collection of that class, the one on which slots exist and in what order as `names(reactives)`, and the one on their number as `length(reactives)`. Unnamed slots show as `reactives$...1`, `reactives$...2` and so on, numbered in the order they were added rather than by position, so a label survives reordering. A read of every slot at once, such as `as.list()`, shows a single dependency on `reactives[]` instead of one per slot. The entry for a slot only appears once it has been read, and the one for the number of slots once `length()` has been called.

Each `shiny::reactiveVal()` that `reactive_vals()`, `$<-`, `[[<-` or `append_value()` creates for a value shows as the label of its slot followed by `()`, `reactive_vals$a()` for slot a and `reactive_vals$...1()` for the first unnamed slot, and the error for calling it once its module is destroyed names it the same way. The label names the slot the value was created for, and stays with the `shiny::reactiveVal()` if that is later bound to another slot.

Examples

```
x <- reactives(a = shiny::reactiveVal(1), b = shiny::reactive(2 * 21))
shiny::isolate(x$a)
shiny::isolate(x[["b"]])

# Assigning a value writes it, adding a slot if there is none
x$a <- 2
x$c <- NULL
shiny::isolate(as_values(x))

# A stored NULL reads as NULL, but `[` finds the slot's reactive
shiny::isolate(is.null(x$c))
shiny::isolate(is.null(x["c"]))
shiny::isolate(has_key(x, c("c", "d")))

# Assigning a reactive with `[<-` binds it, and assigning NULL removes a slot
x["d"] <- shiny::reactive(10 * x$a)
```

```
x["b"] <- NULL
shiny::isolate(names(x))
shiny::isolate(x$d)

# Appending adds an unnamed slot
append_value(x, 3)
append_reactive(x, shiny::reactive(x$a + 1))
shiny::isolate(as_values(x))

y <- reactive_vals(n = 1, label = "one")
shiny::isolate(y$label)

# Everything holding `y` sees the new order
z <- y
reorder(y, c("label", "n"))
shiny::isolate(names(z))

shiny::isolate(as_values(y))
is_reactives(y)

# A value can't be written to a collection of reactive expressions
w <- reactive_exprs(twice = shiny::reactive(2 * y$n))
shiny::isolate(w$twice)
try(w$thrice <- 3)

# A snapshot outlives the session it was taken in
snap <- NULL
shiny::testServer(
  function(input, output, session) {
    x <- reactives(n = shiny::reactive(input$n))
  },
  {
    session$setInputs(n = 21)
    snap <<- snapshot_reactives(x)
  }
)
shiny::isolate(snap$n)
```

Index

append_reactive *(reactives)*, 2
append_value *(reactives)*, 2
as_values *(reactives)*, 2

has_key *(reactives)*, 2

is_reactives *(reactives)*, 2

reactive_exprs *(reactives)*, 2
reactive_vals *(reactives)*, 2
reactives, 2
reorder.reactives *(reactives)*, 2

shiny::debounce(), *2*
shiny::eventReactive(), *2*
shiny::isolate(), *5*
shiny::reactive(), *2–4*
shiny::reactiveVal(), *2–5*
shiny::reactiveValues(), *3, 4*
shiny::reactiveValuesToList(), *3*
shiny::testServer(), *4*
snapshot_reactives *(reactives)*, 2