Package 'rewind'

November 13, 2026

Type Package

Title Undo and Redo for 'shiny' Applications

Version 0.4.0

Description Adds a user-facing undo and redo history to 'shiny' applications. Application state, comprising registered inputs and optionally server-side reactive values, is captured as the user interacts with the application. Users may then step backwards and forwards through that history with the keyboard, with buttons, or by scrubbing a visual history rail. Rapid successive changes, such as dragging a slider, are coalesced into a single history entry, and related changes may be grouped explicitly into semantic steps.

License MIT + file LICENSE

Encoding UTF-8

Language en-GB

Depends R (>= 4.1.0)

Imports htmltools, R6, shiny (>= 1.7.0), utils

Suggests shinytest2, testthat (>= 3.0.0), withr

URL https://github.com/tenmeh/rewind, https://tenmeh.github.io/rewind/

BugReports https://github.com/tenmeh/rewind/issues

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Tanmay Chanda [aut, cre, cph]

Maintainer Tanmay Chanda <tanmaychanda96@gmail.com>

Repository CRAN

Date/Publication 2026-11-13 13:20:08 UTC

Contents

rewind_buttons . . . 2
rewind_dependency . . . 3
rewind_diff . . . 4
rewind_disable . . . 6
rewind_enable . . . 7
rewind_example . . . 10
rewind_exclude . . . 10
rewind_history . . . 11
rewind_pause . . . 12
rewind_status . . . 13
rewind_step . . . 14
rewind_track . . . 16
rewind_ui . . . 17
rewind_undo . . . 18
rewind_unrecorded . . . 19

Index **21**

`rewind_buttons` *Undo and redo buttons*

Description

These two buttons connect to the history of the session. They become enabled and disabled as the history permits. You thus do not have to write an observer on the server.

Usage

```
rewind_buttons(
  undo_label = "Undo",
  redo_label = "Redo",
  class = NULL,
  button_class = NULL
)
```

Arguments

`undo_label`, `redo_label`	The button labels. Use `NULL` for a button with an icon only. The function then sets the accessible name (`aria-label`) to "Undo" or "Redo". A screen reader can thus announce the button correctly when it has no text.
`class`	More CSS classes for the container element.
`button_class`	More CSS classes for the two buttons. Use it for a Bootstrap variant such as `"btn-primary"` or `"btn-outline-secondary"`, and for a size such as `"btn-sm"`.

Value

A htmltools::tagList().

Appearance

The buttons carry `btn btn-default`, so a Bootstrap theme applies to them without any work. Use `button_class` to give them a different Bootstrap variant, size or shape:

```
rewind_buttons(button_class = "btn-primary")
rewind_buttons(button_class = "btn-outline-secondary btn-sm")
```

A variant that you add wins over `btn-default`, so the buttons take the colours of the theme of your application. `bslib::bs_theme()` decides what those colours are. `rewind` thus has no colours of its own to keep in agreement with your application.

Examples

```
rewind_buttons()
rewind_buttons(undo_label = NULL, redo_label = NULL)
rewind_buttons(button_class = "btn-outline-primary btn-sm")
```

---

rewind_dependency *The rewind HTML dependency*

Description

rewind_enable() adds this automatically. You thus rarely need this function. Use it when you want the assets before the server function runs. Use it also when you cannot use `insertUI()`.

Usage

```
rewind_dependency()
```

Value

An htmltools::htmlDependency().

Examples

```
rewind_dependency()
```

`rewind_diff` *Show what changed between two steps*

Description

`rewind_history()` gives one row for each step, with a label such as `"region, year"`. That label says which values changed. This function gives the values themselves: what each one held before, and what it holds now.

Usage

```
rewind_diff(
  from = NULL,
  to = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

`from`, `to`	The two positions to compare, as in the `index` column of `rewind_history()`. The default compares the current position with the one before it. Give both to compare any two positions.
`session`	The Shiny session. The default is the current session.

Details

Use it to tell the user what an undo will do, or to write your own record of what a person changed.

Value

A data frame with one row for each value that differs, in the order of the names. It has seven columns:

- `from`, `to`: the two positions compared.
- `name`: the input ID, or `id$field` for a tracked value.
- `change`: a short description that a person can read.
- `old`, `new`: list columns that hold the values themselves.
- `label`: the name a person reads, from `rewind_enable(labels = )`. It is the same as `name` when there is no label. It is the last column because it was added in 0.4.0, after the other six.

The data frame has no rows when nothing differs, and when there is no earlier step to compare with. It depends on the history, so you can use it in `render*()` and `observe()`.

This is not an audit trail

The history lives in the memory of one session, and it goes when the session goes. It also drops the oldest entries when it passes `depth`, and it drops the steps in front of the position when a change arrives after an undo. A record for an inspection must be written as each change occurs, and kept somewhere else. The history is reactive, so:

```
observeEvent(rewind_history(), {
  write_my_audit_row(rewind_diff())
})
```

Names, and not labels

The `name` column holds the input ID, such as `region`. It does not hold the label of the widget, because Shiny does not give the server the label of an input. A value that `rewind_track()` records appears as `id$field`.

The values can be any type

`old` and `new` are list columns, so they hold the values themselves, whatever their type. The `change` column is a short description for a person to read. It shows both values when they are short, such as `"North -> South"`, and a count when they are not, such as `"12 values"`.

Examples

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

  ui <- fluidPage(
    rewind_buttons(),
    selectInput("region", "Region", c("North", "South", "East")),
    sliderInput("year", "Years", 2018, 2026, c(2018, 2026), sep = ""),
    tableOutput("changed")
  )

  server <- function(input, output, session) {
    rewind_enable()

    # What did the last step change?
    output$changed <- renderTable({
      rewind_diff()[, c("name", "change")]
    })
  }

  shinyApp(ui, server)
}
```

`rewind_disable` *Fully disable undo/redo for a session*

Description

`rewind_pause()` stops capture for a short time. It needs a `rewind_resume()` call after it. This function is different. It removes everything that `rewind_enable()` made:

Usage

```
rewind_disable(session = shiny::getDefaultReactiveDomain())
```

Arguments

`session`	The Shiny session. The default is the current session.

Details

- it destroys each observer;
- it sets the buttons and the history rail in the browser to their empty, disabled condition;
- it returns the session to the condition before `rewind_enable()`.

Call `rewind_enable()` again to start a new history.

Use this function when the application permits undo and redo only in some conditions. An example is a user role that the application does not know at the start of the session.

Value

`TRUE` if the function disabled a session. `FALSE` if `rewind` was not enabled. The function returns the value invisibly.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()
    observeEvent(input$readonly_mode, {
      if (input$readonly_mode) rewind_disable()
    })
  }
}
```

`rewind_enable` *Enable undo and redo for a Shiny session*

Description

Call this once, near the top of your `server` function. After that, `rewind` records the session inputs as the user works. It also records the reactive values that you register with `rewind_track()`. The user can then move backwards and forwards through that history.

Usage

```
rewind_enable(
  session = shiny::getDefaultReactiveDomain(),
  inputs = NULL,
  exclude = NULL,
  depth = 50L,
  coalesce_ms = 400L,
  restore_timeout = 2,
  shortcuts = TRUE,
  verbose = FALSE,
  labels = NULL
)
```

Arguments

`session`	The Shiny session. The default is the current session.
`inputs`	Character vector of the input IDs to capture. Use `NULL` (the default) to capture all permitted inputs.
`exclude`	Character vector of input IDs to skip. `rewind` applies this after `inputs`.
`depth`	The maximum number of history entries to keep. `rewind` removes the oldest entries first.
`coalesce_ms`	The quiet period in milliseconds before `rewind` writes a change to the history. Increase it to group more changes.
`restore_timeout`	The time in seconds that `rewind` waits for the browser to finish a restore. The default of 2 suits a fast connection. Increase it for an application on a slow connection, such as one behind a VPN or on a mobile network. If the limit is too short, capture starts again while the browser is still applying the restore, and a partial state becomes a history entry that the user never made. Do not increase it more than you need. If a restore never returns, `rewind` ignores the changes of the user until this limit ends.
`shortcuts`	Set to `TRUE` to bind the keyboard shortcuts in the browser. There are three: • `Ctrl` + `Z` (`Cmd` + `Z` on macOS) does an undo;

- `Ctrl` + `Shift` + `Z` (`Cmd` + `Shift` + `Z` on macOS) does a redo;
- `Ctrl` + `Y` also does a redo. This is the usual redo shortcut on Windows.

The shortcuts do nothing while the user types in a text field. The text undo of the browser thus continues to work.

`verbose` Set to `TRUE` to show messages about what `rewind` captures and restores. This is useful during development.

`labels` A named character vector of the names a person reads, such as `c(region = "Region", year = "Year range")`. The names are the input ids. A value that `rewind_track()` records is named `id$field`. The labels appear in the history rail, where a step reads `"Region, Year range"` and not `"region, year"`, and in the `label` column of `rewind_diff()`. Shiny does not give the server the label of a widget, so `rewind` cannot find these by itself. An input with no label keeps its id.

Value

The controller, invisibly. Most applications can ignore it.

What gets captured

By default `rewind` captures every input that the page can put back: an input whose element has a Shiny input binding. That covers the inputs of Shiny, and those of other packages that give their widgets a binding.

It leaves out inputs that have no such element. A plot's `hover`, `click`, `dblclick` and `brush`, and the values that DT keeps, such as `_rows_current`, `_rows_all` and `_state`, are sent to the server with `Shiny.setInputValue()` alone. An undo cannot restore them, so recording them made steps that seemed to do nothing. The browser reports which inputs have a binding, including those that `renderUI()` adds later. Until that report arrives, and in `shiny::testServer()`, which has no browser, every input is captured.

To capture an input the page cannot restore, such as the selected rows of a DT table, name it in `inputs`. A list in `inputs` is used as it is. Use `rewind_status()` to see what is captured, and why an input is not.

Five more kinds of input are never captured:

- action buttons and links. Their value is a click counter.
- `shiny::fileInput()`. Its value points to a temporary file on the server. Shiny deletes that file at the next upload. An old snapshot would thus point to a file that does not exist.
- password fields, such as `shiny::passwordInput()`. A password must not be kept in the history, put back in a field by an undo, or shown by `rewind_diff()`. The server cannot tell a password from other text, so the browser reports which fields are passwords, including fields that `renderUI()` adds later. This exclusion applies even when `inputs` names the field. In `shiny::testServer()` there is no browser, so a password field there is captured like any other input.
- inputs with names that start with `rewind_`. These belong to this package.
- inputs with names that start with `.`. These are internal to Shiny.

Text inputs

A word typed at a normal speed is one step. A pause longer than `coalesce_ms` starts a new step, so slow typing can give several. For a text box, `textInput(updateOn = "blur")` gives exactly one step for each edit, when the user leaves the box.

While the cursor is in a text box, `Ctrl + Z` is the browser's own text undo, and `rewind` does nothing (refer to `shortcuts` below). The browser's undo still changes the box, so `rewind` records the change as a new step. To step back through the history, leave the box first, or use the buttons.

Use `inputs` to give a list of the inputs to capture, and `exclude` or `rewind_exclude()` to leave some out. A list is usually better in a large application. Undo must move the controls that the user thinks of as filters. It must not move every other input on the page.

Grouping

Changes that occur within `coalesce_ms` of each other become one history entry. One drag of a slider is thus one undo step, not forty. Use `rewind_step()` to group changes yourself.

Modules

You can call this function inside a `moduleServer()`. `rewind` captures the inputs with their module-local names. These are the same names that `input$` uses inside the module. `rewind` adds the namespace with `session$ns()` when it restores them.

All modules share `session$userData`. A second call to `rewind_enable()` thus uses the same history as the first call. It does not make a second history. Call the function once, at the position in the module tree that is best for your application.

Examples

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

  ui <- fluidPage(
    rewind_buttons(),
    selectInput("species", "Species", c("setosa", "versicolor", "virginica")),
    sliderInput("n", "Rows", 1, 50, 10),
    tableOutput("tbl")
  )

  server <- function(input, output, session) {
    rewind_enable()

    output$tbl <- renderTable({
      head(iris[iris$Species == input$species, ], input$n)
    })
  }

  shinyApp(ui, server)
}
```

rewind_example *Run the demo application*

Description

Starts a small Shiny application that shows `rewind` at work: undo and redo with the buttons and the keyboard, the history rail, and a step that groups several changes. Its source is in the folder `examples/demo` of the installed package.

Usage

```
rewind_example(...)
```

Arguments

`...`	Passed on to `shiny::runApp()`, such as `port` or `launch.browser`.

Value

The value of `shiny::runApp()`, which runs until the application stops.

Examples

```
if (interactive()) {
  rewind_example()
}
```

rewind_exclude *Keep inputs out of the history*

Description

`rewind_enable(exclude = )` leaves inputs out, but only the code that calls `rewind_enable()` can set it. This function can be called from anywhere in the server, a module included, at any time. It works like `shiny::setBookmarkExclude()`: inside a module, give the names that the module uses, and rewind adds the namespace.

Usage

```
rewind_exclude(
  names = character(0),
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

names	Character vector of input names, as the calling module uses them.
session	The Shiny session. The default is the current session.

Details

Two things differ from `setBookmarkExclude()`. A second call adds to the list; it does not replace it, so a module cannot undo an exclusion made elsewhere. And a value already in the history for these inputs is removed, together with any step that changed only them.

A call before `rewind_enable()` in the same session is kept, and applied when `rewind_enable()` runs.

Value

`names`, invisibly.

Examples

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

  search_ui <- function(id) textInput(NS(id, "query"), "Search")
  search_server <- function(id) {
    moduleServer(id, function(input, output, session) {
      # A search box that the user does not expect undo to move.
      rewind_exclude("query")
    })
  }

  ui <- fluidPage(
    rewind_buttons(),
    selectInput("region", "Region", c("North", "South")),
    search_ui("search")
  )

  server <- function(input, output, session) {
    rewind_enable()
    search_server("search")
  }

  shinyApp(ui, server)
}
```

`rewind_history` *Inspect the history*

Description

`rewind_history()` gives one row for each entry that `rewind` keeps. `rewind_can_undo()` and `rewind_can_redo()` tell you if the position can change. All three functions depend on the history. You can thus use them in `render*()` and `observe()` to control your own UI.

Usage

```
rewind_history(session = shiny::getDefaultReactiveDomain())

rewind_can_undo(session = shiny::getDefaultReactiveDomain())

rewind_can_redo(session = shiny::getDefaultReactiveDomain())
```

Arguments

`session`	The Shiny session. The default is the current session.

Value

`rewind_history()` gives a data frame. It has the columns `index`, `label`, `time` and `current`. The other two functions give one logical value.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()
    output$steps <- renderTable(rewind_history())
  }
}
```

---

rewind_pause *Suspend and resume history capture*

Description

While capture is paused, the application applies changes as usual. But `rewind` does not record them. Use this to stop capture for a period that spans more than one reactive flush, such as while the user works in a part of the application that undo must not cover.

Usage

```
rewind_pause(session = shiny::getDefaultReactiveDomain())

rewind_resume(session = shiny::getDefaultReactiveDomain())
```

Arguments

session	The Shiny session. The default is the current session.

Value

TRUE, invisibly.

Not for a block of code

A pause and a resume in the same observer, or the same observeEvent(), have no effect. rewind captures in its own observer, which runs after the one that made the change. By then capture has resumed, so the change is recorded as usual. A change made with update*Input() also comes back from the browser later.

For code that changes the application and must not make a step, such as code that restores a saved view, use rewind_unrecorded().

Examples

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

  server <- function(input, output, session) {
    rewind_enable()
    # Stop capture while an "expert mode" switch is on.
    observeEvent(input$expert_mode, {
      if (input$expert_mode) rewind_pause() else rewind_resume()
    })
  }
}
```

---

rewind_status *See what rewind captures, and why it leaves an input out*

Description

An undo that skips an input is hard to explain from outside. The input can be excluded, can be a password field, can be missing from inputs, or can be one that the page cannot put back. This function lists every input of the session with the reason, so one call answers the question.

Usage

```
rewind_status(session = shiny::getDefaultReactiveDomain())
```

Arguments

session	The Shiny session. The default is the current session.

Details

It reads every input, so inside `render*()` the result follows the inputs and the history. That makes it useful in a debug panel, for example `renderPrint(rewind_status())`.

Value

A list:

- `captured`: the names of the inputs that rewind records.
- `skipped`: a named character vector. Each name is an input, and each value is the reason it is not recorded: `"excluded"`, `"password field"`, `"not in inputs"`, `"cannot be restored by the page"`, `"action button"`, `"file upload"` or `"NULL value"`. Inputs internal to Shiny and to rewind are left out of the list.
- `tracked`: the groups registered with `rewind_track()`, each with the fields it names (`NULL` for all fields).
- `restorable`: the inputs that the page reported it can restore, or `NULL` before the report arrives. Refer to `rewind_enable()`.
- `paused`: `TRUE` between `rewind_pause()` and `rewind_resume()`.
- `size`, `position`: the number of steps, and the current one.

Examples

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

  ui <- fluidPage(
    rewind_buttons(),
    selectInput("region", "Region", c("North", "South")),
    passwordInput("pw", "Password"),
    verbatimTextOutput("status")
  )

  server <- function(input, output, session) {
    rewind_enable()
    output$status <- renderPrint(rewind_status()$skipped)
  }

  shinyApp(ui, server)
}
```

---

rewind_step *Group several changes into one undo step*

Description

This function holds a block of code. Every change in that block becomes one history entry with the label that you give. Use it for buttons such as "reset all filters" or "apply preset". For these buttons, the standard time-based grouping can make several steps, or it can give a label that does not help the user.

Usage

```
rewind_step(
  expr,
  label = NULL,
  hold_ms = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

expr	The code to run. This is usually a set of `update*Input()` calls, or assignments to tracked reactive values.
label	The label for the history entry.
hold_ms	The time in milliseconds to keep the entry open. The default is two times the `coalesce_ms` of the session. Increase it if a slow browser makes two steps from one block.
session	The Shiny session. The default is the current session.

Details

The block runs immediately. `rewind` writes the history entry after the changes return from the browser.

Value

The value of `expr`, invisibly.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()

    observeEvent(input$reset, {
      rewind_step(label = "Reset filters", {
        updateSelectInput(session, "region", selected = "All")
        updateSliderInput(session, "year", value = c(2000, 2026))
        updateCheckboxInput(session, "only_active", value = FALSE)
      })
    })
  }
}
```

`rewind_track` *Include server-side reactive values in the history*

Description

`rewind` captures inputs automatically. It cannot see the state that you keep in a `shiny::reactiveValues()` object. Register that object here. `rewind` then records the registered fields with the inputs, and writes them back at an undo.

Usage

```
rewind_track(
  values,
  fields = NULL,
  id = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

`values`	A `shiny::reactiveValues()` object.
`fields`	Character vector of the field names to track. Use `NULL` for all the fields that exist when `rewind` takes the snapshot.
`id`	A name for this group. `rewind` uses it to keep the registered objects apart, and to label the history entries. The default is the name of the `values` argument at the call.
`session`	The Shiny session. The default is the current session.

Details

Register only the values that are true *state*. A derived value or a cached value does no harm, but it has no use. Do not register a value that an observer computes again immediately. The undo step then appears to do nothing.

Value

`TRUE`, invisibly.

Environments and R6 objects

`rewind` keeps each value as it is, and does not copy it. For a vector, a list or a data frame, that is a true record, because R copies such a value when you change it.

An environment and an R6 object are different. The history holds the object itself, not a copy. A change inside the object thus has two effects:

- It is not a step. When you assign the same object back, Shiny sees no change, so `rewind` sees none.

- An undo cannot bring back the old contents. Each entry holds the same object, as it is now.

To make each change a step, change a copy, then assign the copy:

```
model <- state$model$clone(deep = TRUE)
model$fit(data)
state$model <- model
```

For an environment, keep a list in the reactive values instead.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()
    state <- reactiveValues(selected = character(), zoom = 1)
    rewind_track(state, fields = c("selected", "zoom"))
  }
}
```

`rewind_ui` *A scrubbable history rail*

Description

This function draws the history as a vertical list of steps. The newest step is at the end. The rail shows the current position. A click on a step moves to that step. The rail gets its data from the server. You thus do not have to write a `render` function.

Usage

```
rewind_ui(label = "History", max_height = "20rem", class = NULL)
```

Arguments

`label`	The heading above the rail. Use `NULL` for no heading.
`max_height`	The CSS height at which the rail starts to scroll.
`class`	More CSS classes for the container element.

Value

A `htmltools::tagList()`.

Examples

```
rewind_ui()
rewind_ui(label = "Steps", max_height = "12rem")
```

rewind_undo *Move through the history programmatically*

Description

The keyboard shortcuts and rewind_buttons() are sufficient for most applications. Use these functions to move through the history from your own controls.

Usage

```
rewind_undo(session = shiny::getDefaultReactiveDomain())

rewind_redo(session = shiny::getDefaultReactiveDomain())

rewind_jump(index, session = shiny::getDefaultReactiveDomain())

rewind_clear(session = shiny::getDefaultReactiveDomain())
```

Arguments

session	The Shiny session. The default is the current session.
index	The position to move to. The first position is 1. Position 1 holds the oldest entry that rewind keeps.

Value

TRUE if the position changed. If not, FALSE. The functions return the value invisibly.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()
    observeEvent(input$my_back_button, rewind_undo())
  }
}
```

`rewind_unrecorded` *Make changes that are not a step*

Description

This function holds a block of code. The changes that the block makes do not become a step. They become part of the current state instead, so an undo never goes back across them. Use it for code that sets up a view: a saved view that you restore, a bookmark that you apply, or defaults that you load when the user logs in.

Usage

```
rewind_unrecorded(
  expr,
  hold_ms = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

`expr`	The code to run. This is usually a set of `update*Input()` calls, or assignments to tracked reactive values.
`hold_ms`	The time in milliseconds to keep the window open after the block. The default is two times the `coalesce_ms` of the session. Increase it if a slow browser lets a change through as a step.
`session`	The Shiny session. The default is the current session.

Details

It is the other side of `rewind_step()`. That function makes the changes of a block into one step with a label. This one makes them into no step at all.

Value

The value of `expr`, invisibly.

Why not rewind_pause() and rewind_resume()

A pause and a resume in one observer have no effect. `rewind` captures in its own observer, which runs after the one that made the change, so capture has resumed by then. This function keeps a window open after the block instead, so that the changes that come back from the browser later are covered too.

What happens around it

- A change from the user that still waits to be written becomes its own step first, so the block does not swallow it.
- The steps in front of the position go, as they do for any change.
- A change that the user makes inside the window is part of the current state too. Keep `hold_ms` short.

Examples

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

  server <- function(input, output, session) {
    rewind_enable()

    # A saved view. Loading it is not something to undo.
    observeEvent(input$load_view, {
      rewind_unrecorded({
        updateSelectInput(session, "region", selected = "South")
        updateSliderInput(session, "year", value = c(2020, 2024))
      })
    })
  }
}
```

Index

htmltools::htmlDependency(), *3*
htmltools::tagList(), *3*, *17*

rewind_buttons, 2
rewind_buttons(), *18*
rewind_can_redo *(*rewind_history*)*, 11
rewind_can_undo *(*rewind_history*)*, 11
rewind_clear *(*rewind_undo*)*, 18
rewind_dependency, 3
rewind_diff, 4
rewind_diff(), *8*
rewind_disable, 6
rewind_enable, 7
rewind_enable(), *3*, *6*, *11*, *14*
rewind_example, 10
rewind_exclude, 10
rewind_exclude(), *9*
rewind_history, 11
rewind_history(), *4*
rewind_jump *(*rewind_undo*)*, 18
rewind_pause, 12
rewind_pause(), *6*, *14*
rewind_redo *(*rewind_undo*)*, 18
rewind_resume *(*rewind_pause*)*, 12
rewind_resume(), *6*, *14*
rewind_status, 13
rewind_status(), *8*
rewind_step, 14
rewind_step(), *9*, *19*
rewind_track, 16
rewind_track(), *5*, *7*, *8*, *14*
rewind_ui, 17
rewind_undo, 18
rewind_unrecorded, 19
rewind_unrecorded(), *13*

shiny::fileInput(), *8*
shiny::passwordInput(), *8*
shiny::reactiveValues(), *16*
shiny::runApp(), *10*
shiny::setBookmarkExclude(), *10*
shiny::testServer(), *8*