Package 'shiny'

August 27, 2013

Type Package

Title Web Application Framework for R

Version 0.7.0

Date 2013-01-23

Author RStudio, Inc.

Maintainer Winston Chang <winston@rstudio.com>

Description Shiny makes it incredibly easy to build interactive web applications with R. Automatic ''reactive'' binding between inputs and outputs and extensive pre-built widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.

License GPL-3

Depends R (>= 2.14.1)

Imports stats, tools, utils, methods, httpuv (>= 1.1.0), caTools,RJSONIO, xtable, digest

Suggests datasets, markdown, Cairo, testthat

URL http://www.rstudio.com/shiny/

BugReports https://github.com/rstudio/shiny/issues

Collate
'map.R' 'priorityqueue.R' 'utils.R' 'tar.R' 'timer.R' 'tags.R''cache.R' 'graph.R' 'react.R' 'reactives.R' 'fileupload.R''shiny
url.R' 'imageutils.R' 'update-input.R'

NeedsCompilation no

Repository CRAN

Date/Publication 2013-08-27 17:21:04

R topics documented:

shiny-package *Web Application Framework for R*

Description

Shiny makes it incredibly easy to build interactive web applications with R. Automatic "reactive" binding between inputs and outputs and extensive pre-built widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.

Details

The Shiny tutorial at http://rstudio.github.com/shiny/tutorial explains the framework in depth, walks you through building a simple application, and includes extensive annotated examples.

actionButton *Action button*

Description

Creates an action button whose value is initially zero, and increments by one each time it is pressed.

Usage

```
actionButton(inputId, label)
```

Arguments

inputId	Specifies the input slot that will be used to access the value.
label	The contents of the button–usually a text label, but you could also use any other HTML, like an image.

addResourcePath *Resource Publishing*

Description

Adds a directory of static resources to Shiny's web server, with the given path prefix. Primarily intended for package authors to make supporting JavaScript/CSS files available to their components.

Usage

```
addResourcePath(prefix, directoryPath)
```

Arguments

prefix	The URL prefix (without slashes). Valid characters are a-z, A-Z, 0-9, hyphen, and underscore; and must begin with a-z or A-Z. For example, a value of 'foo' means that any request paths that begin with '/foo' will be mapped to the given directory.
directoryPath	The directory that contains the static resources to be served.

Details

You can call addResourcePath multiple times for a given prefix; only the most recent value will be retained. If the normalized directoryPath is different than the directory that's currently mapped to the prefix, a warning will be issued.

See Also

singleton

Examples

```
addResourcePath('datasets', system.file('data', package='datasets'))
```

---

animationOptions *Animation Options*

Description

Creates an options object for customizing animations for sliderInput.

Usage

```
animationOptions(interval = 1000, loop = FALSE,
  playButton = NULL, pauseButton = NULL)
```

Arguments

interval	The interval, in milliseconds, between each animation step.
loop	TRUE to automatically restart the animation when it reaches the end.
playButton	Specifies the appearance of the play button. Valid values are a one-element character vector (for a simple text label), an HTML tag or list of tags (using tag and friends), or raw HTML (using HTML).
pauseButton	Similar to playButton, but for the pause button.

bootstrapPage *Create a Twitter Bootstrap page*

Description

Create a Shiny UI page that loads the CSS and JavaScript for Twitter Bootstrap, and has no content in the page body (other than what you provide).

Usage

```
bootstrapPage(...)

basicPage(...)
```

Arguments

...	The contents of the document body.

Details

These functions are primarily intended for users who are proficient in HTML/CSS, and know how to lay out pages in Bootstrap. Most users should use template functions like `pageWithSidebar`.

`basicPage` is the same as `bootstrapPage`, with an added `<div class="container-fluid">` wrapper to provide a little padding.

Value

A UI defintion that can be passed to the shinyUI function.

builder *HTML Builder Functions*

Description

Simple functions for constructing HTML documents.

Usage

```
p(...)
h1(...)
h2(...)
h3(...)
h4(...)
h5(...)
h6(...)
a(...)
```

```
br(...)
div(...)
span(...)
pre(...)
code(...)
img(...)
strong(...)
em(...)

tags
```

Arguments

...	Attributes and children of the element. Named arguments become attributes, and positional arguments become children. Valid children are tags, single-character character vectors (which become text nodes), and raw HTML (see HTML). You can also pass lists that contain tags, text nodes, and HTML.

Details

The tags environment contains convenience functions for all valid HTML5 tags. To generate tags that are not part of the HTML5 specification, you can use the tag function.

Dedicated functions are available for the most common HTML tags that do not conflict with common R functions.

The result from these functions is a tag object, which can be converted using as.character.

Examples

```
doc <- tags$html(
  tags$head(
    tags$title('My first page')
  ),
  tags$body(
    h1('My first heading'),
    p('My first paragraph, with some ',
      strong('bold'),
      ' text.'),
    div(id='myDiv', class='simpleDiv',
        'Here is a div with some attributes.')
  )
)
cat(as.character(doc))
```

---

checkboxGroupInput *Checkbox Group Input Control*

Description

Create a group of checkboxes that can be used to toggle multiple choices independently. The server will receive the input as a character vector of the selected values.

Usage

```
checkboxGroupInput(inputId, label, choices,
  selected = NULL)
```

Arguments

inputId	Input variable to assign the control's value to.
label	Display label for the control.
choices	List of values to show checkboxes for. If elements of the list are named then that name rather than the value is displayed to the user.
selected	Names of items that should be initially selected, if any.

Value

A list of HTML elements that can be added to a UI definition.

See Also

checkboxInput, updateCheckboxGroupInput

Examples

```
checkboxGroupInput("variable", "Variable:",
                   c("Cylinders" = "cyl",
                     "Transmission" = "am",
                     "Gears" = "gear"))
```

---

checkboxInput *Checkbox Input Control*

Description

Create a checkbox that can be used to specify logical values.

Usage

```
checkboxInput(inputId, label, value = FALSE)
```

Arguments

inputId	Input variable to assign the control's value to.
label	Display label for the control.
value	Initial value (TRUE or FALSE).

Value

A checkbox control that can be added to a UI definition.

See Also

checkboxGroupInput, updateCheckboxInput

Examples

```
checkboxInput("outliers", "Show outliers", FALSE)
```

`conditionalPanel` *Conditional Panel*

Description

Creates a panel that is visible or not, depending on the value of a JavaScript expression. The JS expression is evaluated once at startup and whenever Shiny detects a relevant change in input/output.

Usage

```
conditionalPanel(condition, ...)
```

Arguments

`condition`	A JavaScript expression that will be evaluated repeatedly to determine whether the panel should be displayed.
`...`	Elements to include in the panel.

Details

In the JS expression, you can refer to `input` and `output` JavaScript objects that contain the current values of input and output. For example, if you have an input with an id of `foo`, then you can use `input.foo` to read its value. (Be sure not to modify the input/output objects, as this may cause unpredictable behavior.)

Examples

```
sidebarPanel(
   selectInput(
      "plotType", "Plot Type",
      c(Scatter = "scatter",
        Histogram = "hist")),

   # Only show this panel if the plot type is a histogram
   conditionalPanel(
      condition = "input.plotType == 'hist'",
      selectInput(
```

```
          "breaks", "Breaks",
          c("Sturges",
            "Scott",
            "Freedman-Diaconis",
            "[Custom]" = "custom")),

        # Only show this panel if Custom is selected
        conditionalPanel(
          condition = "input.breaks == 'custom'",
          sliderInput("breakCount", "Break Count", min=1, max=1000, value=10)
        )
      )
)
```

`dateInput` *Create date input*

Description

Creates a text input which, when clicked on, brings up a calendar that the user can click on to select dates.

Usage

```
dateInput(inputId, label, value = NULL, min = NULL,
  max = NULL, format = "yyyy-mm-dd", startview = "month",
  weekstart = 0, language = "en")
```

Arguments

`inputId`	Input variable to assign the control's value to.
`label`	Display label for the control.
`value`	The starting date. Either a Date object, or a string in `yyyy-mm-dd` format. If NULL (the default), will use the current date in the client's time zone.
`min`	The minimum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`max`	The maximum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`format`	The format of the date to display in the browser. Defaults to `"yyyy-mm-dd"`.
`startview`	The date range shown when the input object is first clicked. Can be "month" (the default), "year", or "decade".
`weekstart`	Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).
`language`	The language used for month and day names. Default is "en". Other valid values include "bg", "ca", "cs", "da", "de", "el", "es", "fi", "fr", "he", "hr", "hu", "id", "is", "it", "ja", "kr", "lt", "lv", "ms", "nb", "nl", "pl", "pt", "pt", "ro", "rs", "rs-latin", "ru", "sk", "sl", "sv", "sw", "th", "tr", "uk", "zh-CN", and "zh-TW".

Details

The date `format` string specifies how the date will be displayed in the browser. It allows the following values:

- `yy` Year without century (12)
- `yyyy` Year with century (2012)
- `mm` Month number, with leading zero (01-12)
- `m` Month number, without leading zero (01-12)
- `M` Abbreviated month name
- `MM` Full month name
- `dd` Day of month with leading zero
- `d` Day of month without leading zero
- `D` Abbreviated weekday name
- `DD` Full weekday name

See Also

`dateRangeInput`, `updateDateInput`

Examples

```
dateInput("date", "Date:", value = "2012-02-29")

# Default value is the date in client's time zone
dateInput("date", "Date:")

# value is always yyyy-mm-dd, even if the display format is different
dateInput("date", "Date:", value = "2012-02-29", format = "mm/dd/yy")

# Pass in a Date object
dateInput("date", "Date:", value = Sys.Date()-10)

# Use different language and different first day of week
dateInput("date", "Date:",
          language = "de",
          weekstart = 1)

# Start with decade view instead of default month view
dateInput("date", "Date:",
          startview = "decade")
```

`dateRangeInput` *Create date range input*

Description

Creates a pair of text inputs which, when clicked on, bring up calendars that the user can click on to select dates.

Usage

```
dateRangeInput(inputId, label, start = NULL, end = NULL,
  min = NULL, max = NULL, format = "yyyy-mm-dd",
  startview = "month", weekstart = 0, language = "en",
  separator = " to ")
```

Arguments

`inputId`	Input variable to assign the control's value to.
`label`	Display label for the control.
`start`	The initial start date. Either a Date object, or a string in `yyyy-mm-dd` format. If NULL (the default), will use the current date in the client's time zone.
`end`	The initial end date. Either a Date object, or a string in `yyyy-mm-dd` format. If NULL (the default), will use the current date in the client's time zone.
`min`	The minimum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`max`	The maximum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`format`	The format of the date to display in the browser. Defaults to `"yyyy-mm-dd"`.
`startview`	The date range shown when the input object is first clicked. Can be "month" (the default), "year", or "decade".
`weekstart`	Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).
`language`	The language used for month and day names. Default is "en". Other valid values include "bg", "ca", "cs", "da", "de", "el", "es", "fi", "fr", "he", "hr", "hu", "id", "is", "it", "ja", "kr", "lt", "lv", "ms", "nb", "nl", "pl", "pt", "pt", "ro", "rs", "rs-latin", "ru", "sk", "sl", "sv", "sw", "th", "tr", "uk", "zh-CN", and "zh-TW".
`separator`	String to display between the start and end input boxes.

Details

The date `format` string specifies how the date will be displayed in the browser. It allows the following values:

- `yy` Year without century (12)

- yyyy Year with century (2012)
- mm Month number, with leading zero (01-12)
- m Month number, without leading zero (01-12)
- M Abbreviated month name
- MM Full month name
- dd Day of month with leading zero
- d Day of month without leading zero
- D Abbreviated weekday name
- DD Full weekday name

See Also

dateInput, updateDateRangeInput

Examples

```
dateRangeInput("daterange", "Date range:",
               start = "2001-01-01",
               end   = "2010-12-31")

# Default start and end is the current date in the client's time zone
dateRangeInput("daterange", "Date range:")

# start and end are always specified in yyyy-mm-dd, even if the display
# format is different
dateRangeInput("daterange", "Date range:",
               start  = "2001-01-01",
               end    = "2010-12-31",
               min    = "2001-01-01",
               max    = "2012-12-21",
               format = "mm/dd/yy",
               separator = " - ")

# Pass in Date objects
dateRangeInput("daterange", "Date range:",
               start = Sys.Date()-10,
               end = Sys.Date()+10)

# Use different language and different first day of week
dateRangeInput("daterange", "Date range:",
               language = "de",
               weekstart = 1)

# Start with decade view instead of default month view
dateRangeInput("daterange", "Date range:",
               startview = "decade")
```

`downloadButton` *Create a download button or link*

Description

Use these functions to create a download button or link; when clicked, it will initiate a browser download. The filename and contents are specified by the corresponding `downloadHandler` defined in the server function.

Usage

```
downloadButton(outputId, label = "Download",
  class = NULL)

downloadLink(outputId, label = "Download", class = NULL)
```

Arguments

`outputId`	The name of the output slot that the `downloadHandler` is assigned to.
`label`	The label that should appear on the button.
`class`	Additional CSS classes to apply to the tag, if any.

See Also

downloadHandler

Examples

```
## Not run:
# In server.R:
output$downloadData <- downloadHandler(
  filename = function() {
    paste('data-', Sys.Date(), '.csv', sep='')
  },
  content = function(con) {
    write.csv(data, con)
  }
)

# In ui.R:
downloadLink('downloadData', 'Download')

## End(Not run)
```

`downloadHandler` *File Downloads*

Description

Allows content from the Shiny application to be made available to the user as file downloads (for example, downloading the currently visible data as a CSV file). Both filename and contents can be calculated dynamically at the time the user initiates the download. Assign the return value to a slot on `output` in your server function, and in the UI use `downloadButton` or `downloadLink` to make the download available.

Usage

```
downloadHandler(filename, content, contentType = NA)
```

Arguments

`filename`	A string of the filename, including extension, that the user's web browser should default to when downloading the file; or a function that returns such a string. (Reactive values and functions may be used from this function.)
`content`	A function that takes a single argument `file` that is a file path (string) of a nonexistent temp file, and writes the content to that file path. (Reactive values and functions may be used from this function.)
`contentType`	A string of the download's content type, for example `"text/csv"` or `"image/png"`. If `NULL` or `NA`, the content type will be guessed based on the filename extension, or `application/octet-stream` if the extension is unknown.

Examples

```
## Not run:
# In server.R:
output$downloadData <- downloadHandler(
  filename = function() {
    paste('data-', Sys.Date(), '.csv', sep='')
  },
  content = function(file) {
    write.csv(data, file)
  }
)

# In ui.R:
downloadLink('downloadData', 'Download')

## End(Not run)
```

exprToFunction *Convert an expression or quoted expression to a function*

Description

This is to be called from another function, because it will attempt to get an unquoted expression from two calls back.

Usage

```
exprToFunction(expr, env = parent.frame(2),
  quoted = FALSE)
```

Arguments

expr	A quoted or unquoted expression, or a function.
env	The desired environment for the function. Defaults to the calling environment two steps back.
quoted	Is the expression quoted?

Details

If expr is a quoted expression, then this just converts it to a function. If expr is a function, then this simply returns expr (and prints a deprecation message. If expr was a non-quoted expression from two calls back, then this will quote the original expression and convert it to a function.

Examples

```
# Example of a new renderer, similar to renderText
# This is something that toolkit authors will do
renderTriple <- function(expr, env=parent.frame(), quoted=FALSE) {
  # Convert expr to a function
  func <- shiny::exprToFunction(expr, env, quoted)

  function() {
    value <- func()
    paste(rep(value, 3), collapse=", ")
  }
}


# Example of using the renderer.
# This is something that app authors will do.
values <- reactiveValues(A="text")

## Not run:
# Create an output object
output$tripleA <- renderTriple({
```

```
  values$A
})

## End(Not run)

# At the R console, you can experiment with the renderer using isolate()
tripleA <- renderTriple({
  values$A
})

isolate(tripleA())
# "text, text, text"
```

`fileInput` *File Upload Control*

Description

Create a file upload control that can be used to upload one or more files. **Does not work on older browsers, including Internet Explorer 9 and earlier.**

Usage

```
fileInput(inputId, label, multiple = FALSE,
  accept = NULL)
```

Arguments

`inputId`	Input variable to assign the control's value to.
`label`	Display label for the control.
`multiple`	Whether the user should be allowed to select and upload multiple files at once.
`accept`	A character vector of MIME types; gives the browser a hint of what kind of files the server is expecting.

Details

Whenever a file upload completes, the corresponding input variable is set to a dataframe. This dataframe contains one row for each selected file, and the following columns:

`name` The filename provided by the web browser. This is **not** the path to read to get at the actual data that was uploaded (see `datapath` column).

`size` The size of the uploaded data, in bytes.

`type` The MIME type reported by the browser (for example, `text/plain`), or empty string if the browser didn't know.

`datapath` The path to a temp file that contains the data that was uploaded. This file may be deleted if the user performs another upload operation.

headerPanel *Create a header panel*

Description

Create a header panel containing an application title.

Usage

```
headerPanel(title, windowTitle = title)
```

Arguments

title	An application title to display
windowTitle	The title that should be displayed by the browser window. Useful if title is not a string.

Value

A headerPanel that can be passed to pageWithSidebar

Examples

```
headerPanel("Hello Shiny!")
```

helpText *Create a help text element*

Description

Create help text which can be added to an input form to provide additional explanation or context.

Usage

```
helpText(...)
```

Arguments

...	One or more help text strings (or other inline HTML elements)

Value

A help text element that can be added to a UI definition.

Examples

```
helpText("Note: while the data view will show only",
         "the specified number of observations, the",
         "summary will be based on the full dataset.")
```

HTML *Mark Characters as HTML*

Description

Marks the given text as HTML, which means the tag functions will know not to perform HTML escaping on it.

Usage

```
HTML(text, ...)
```

Arguments

`text`	The text value to mark with HTML
`...`	Any additional values to be converted to character and concatenated together

Value

The same value, but marked as HTML.

Examples

```
el <- div(HTML("I like <u>turtles</u>"))
cat(as.character(el))
```

---

htmlOutput *Create an HTML output element*

Description

Render a reactive output variable as HTML within an application page. The text will be included within an HTML `div` tag, and is presumed to contain HTML content which should not be escaped.

Usage

```
htmlOutput(outputId)

uiOutput(outputId)
```

Arguments

outputId	output variable to read the value from

Details

uiOutput is intended to be used with renderUI on the server side. It is currently just an alias for htmlOutput.

Value

An HTML output element that can be included in a panel

Examples

```
htmlOutput("summary")
```

---

imageOutput *Create a image output element*

Description

Render a renderImage within an application page.

Usage

```
imageOutput(outputId, width = "100%", height = "400px")
```

Arguments

outputId	output variable to read the image from
width	Image width. Must be a valid CSS unit (like "100%", "400px", "auto") or a number, which will be coerced to a string and have "px" appended.
height	Image height

Value

An image output element that can be included in a panel

Examples

```
# Show an image
mainPanel(
  imageOutput("dataImage")
)
```

includeHTML *Include Content From a File*

Description

Include HTML, text, or rendered Markdown into a Shiny UI.

Usage

```
includeHTML(path)

includeText(path)

includeMarkdown(path)

includeCSS(path, ...)

includeScript(path, ...)
```

Arguments

path	The path of the file to be included. It is highly recommended to use a relative path (the base path being the Shiny application directory), not an absolute path.
...	Any additional attributes to be applied to the generated tag.

Details

These functions provide a convenient way to include an extensive amount of HTML, textual, Markdown, CSS, or JavaScript content, rather than using a large literal R string.

Note

`includeText` escapes its contents, but does no other processing. This means that hard breaks and multiple spaces will be rendered as they usually are in HTML: as a single space character. If you are looking for preformatted text, wrap the call with `pre`, or consider using `includeMarkdown` instead.

The `includeMarkdown` function requires the `markdown` package.

invalidateLater *Scheduled Invalidation*

Description

Schedules the current reactive context to be invalidated in the given number of milliseconds.

Usage

```
invalidateLater(millis, session)
```

Arguments

millis	Approximate milliseconds to wait before invalidating the current reactive context.
session	A session object. This is needed to cancel any scheduled invalidations after a user has ended the session. If NULL, then this invalidation will not be tied to any session, and so it will still occur.

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # Re-execute this reactive expression after 1000 milliseconds
    invalidateLater(1000, session)

    # Do something each time this is invalidated.
    # The isolate() makes this observer _not_ get invalidated and re-executed
    # when input$n changes.
    print(paste("The value of input$n is", isolate(input$n)))
  })

  # Generate a new histogram at timed intervals, but not when
  # input$n changes.
  output$plot <- renderPlot({
    # Re-execute this reactive expression after 2000 milliseconds
    invalidateLater(2000, session)
    hist(isolate(input$n))
  })
})

## End(Not run)
```

---

is.reactivevalues *Checks whether an object is a reactivevalues object*

Description

Checks whether its argument is a reactivevalues object.

Usage

```
is.reactivevalues(x)
```

Arguments

x	The object to test.

See Also

reactiveValues.

---

isolate *Create a non-reactive scope for an expression*

Description

Executes the given expression in a scope where reactive values or expression can be read, but they cannot cause the reactive scope of the caller to be re-evaluated when they change.

Usage

```
isolate(expr)
```

Arguments

expr	An expression that can access reactive values or expressions.

Details

Ordinarily, the simple act of reading a reactive value causes a relationship to be established between the caller and the reactive value, where a change to the reactive value will cause the caller to re-execute. (The same applies for the act of getting a reactive expression's value.) The `isolate` function lets you read a reactive value or expression without establishing this relationship.

The expression given to `isolate()` is evaluated in the calling environment. This means that if you assign a variable inside the `isolate()`, its value will be visible outside of the `isolate()`. If you want to avoid this, you can use `local()` inside the `isolate()`.

This function can also be useful for calling reactive expression at the console, which can be useful for debugging. To do so, simply wrap the calls to the reactive expression with `isolate()`.

Examples

```
## Not run:
observe({
  input$saveButton  # Do take a dependency on input$saveButton

  # isolate a simple expression
  data <- get(isolate(input$dataset))  # No dependency on input$dataset
  writeToDatabase(data)
})

observe({
```

```
  input$saveButton  # Do take a dependency on input$saveButton

  # isolate a whole block
  data <- isolate({
    a <- input$valueA   # No dependency on input$valueA or input$valueB
    b <- input$valueB
    c(a=a, b=b)
  })
  writeToDatabase(data)
})

observe({
  x <- 1
  # x outside of isolate() is affected
  isolate(x <- 2)
  print(x) # 2

  y <- 1
  # Use local() to avoid affecting calling environment
  isolate(local(y <- 2))
  print(y) # 1
})


## End(Not run)

# Can also use isolate to call reactive expressions from the R console
values <- reactiveValues(A=1)
fun <- reactive({ as.character(values$A) })
isolate(fun())
# "1"

# isolate also works if the reactive expression accesses values from the
# input object, like input$x
```

`mainPanel` *Create a main panel*

Description

Create a main panel containing output elements that can in turn be passed to pageWithSidebar.

Usage

```
mainPanel(...)
```

Arguments

`...`	Ouput elements to include in the main panel

Value

A main panel that can be passed to pageWithSidebar

Examples

```
# Show the caption and plot of the requested variable against mpg
mainPanel(
   h3(textOutput("caption")),
   plotOutput("mpgPlot")
)
```

`numericInput` *Create a numeric input control*

Description

Create an input control for entry of numeric values

Usage

```
numericInput(inputId, label, value, min = NA, max = NA,
  step = NA)
```

Arguments

`inputId`	Input variable to assign the control's value to
`label`	Display label for the control
`value`	Initial value
`min`	Minimum allowed value
`max`	Maximum allowed value
`step`	Interval to use when stepping between min and max

Value

A numeric input control that can be added to a UI definition.

See Also

updateNumericInput

Examples

```
numericInput("obs", "Observations:", 10,
             min = 1, max = 100)
```

`observe` *Create a reactive observer*

Description

Creates an observer from the given expression.

Usage

```
observe(x, env = parent.frame(), quoted = FALSE,
  label = NULL, suspended = FALSE, priority = 0)
```

Arguments

`x`	An expression (quoted or unquoted). Any return value will be ignored.
`env`	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression.
`quoted`	Is the expression quoted? By default, this is `FALSE`. This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with 'quote()'.
`label`	A label for the observer, useful for debugging.
`suspended`	If `TRUE`, start the observer in a suspended state. If `FALSE` (the default), start in a non-suspended state.
`priority`	An integer or numeric that controls the priority with which this observer should be executed. An observer with a given priority level will always execute sooner than all observers with a lower priority level. Positive, negative, and zero values are allowed.

Details

An observer is like a reactive expression in that it can read reactive values and call reactive expressions, and will automatically re-execute when those dependencies change. But unlike reactive expressions, it doesn't yield a result and can't be used as an input to other reactive expressions. Thus, observers are only useful for their side effects (for example, performing I/O).

Another contrast between reactive expressions and observers is their execution strategy. Reactive expressions use lazy evaluation; that is, when their dependencies change, they don't re-execute right away but rather wait until they are called by someone else. Indeed, if they are not called then they will never re-execute. In contrast, observers use eager evaluation; as soon as their dependencies change, they schedule themselves to re-execute.

Value

An observer reference class object. This object has the following methods:

`suspend()` Causes this observer to stop scheduling flushes (re-executions) in response to invalidations. If the observer was invalidated prior to this call but it has not re-executed yet then that re-execution will still occur, because the flush is already scheduled.

`resume()` Causes this observer to start re-executing in response to invalidations. If the observer was invalidated while suspended, then it will schedule itself for re-execution.

`setPriority(priority = 0)` Change this observer's priority. Note that if the observer is currently invalidated, then the change in priority will not take effect until the next invalidation–unless the observer is also currently suspended, in which case the priority change will be effective upon resume.

`onInvalidate(callback)` Register a callback function to run when this observer is invalidated. No arguments will be provided to the callback function when it is invoked.

Examples

```
values <- reactiveValues(A=1)

obsB <- observe({
  print(values$A + 1)
})

# Can use quoted expressions
obsC <- observe(quote({ print(values$A + 2) }), quoted = TRUE)

# To store expressions for later conversion to observe, use quote()
expr_q <- quote({ print(values$A + 3) })
obsD <- observe(expr_q, quoted = TRUE)

# In a normal Shiny app, the web client will trigger flush events. If you
# are at the console, you can force a flush with flushReact()
shiny:::flushReact()
```

`outputOptions` *Set options for an output object.*

Description

These are the available options for an output object:

- suspendWhenHidden. When `TRUE` (the default), the output object will be suspended (not execute) when it is hidden on the web page. When `FALSE`, the output object will not suspend when hidden, and if it was already hidden and suspended, then it will resume immediately.
- priority. The priority level of the output object. Queued outputs with higher priority values will execute before those with lower values.

Usage

```
outputOptions(x, name, ...)
```

Arguments

x	A shinyoutput object (typically output).
name	The name of an output observer in the shinyoutput object.
...	Options to set for the output observer.

Examples

```
## Not run:
# Get the list of options for all observers within output
outputOptions(output)

# Disable suspend for output$myplot
outputOptions(output, "myplot", suspendWhenHidden = FALSE)

# Change priority for output$myplot
outputOptions(output, "myplot", priority = 10)

# Get the list of options for output$myplot
outputOptions(output, "myplot")

## End(Not run)
```

---

pageWithSidebar *Create a page with a sidebar*

Description

Create a Shiny UI that contains a header with the application title, a sidebar for input controls, and a main area for output.

Usage

```
pageWithSidebar(headerPanel, sidebarPanel, mainPanel)
```

Arguments

headerPanel	The headerPanel with the application title
sidebarPanel	The sidebarPanel containing input controls
mainPanel	The mainPanel containing outputs

Value

A UI defintion that can be passed to the shinyUI function

Examples

```
# Define UI
shinyUI(pageWithSidebar(

  # Application title
  headerPanel("Hello Shiny!"),

  # Sidebar with a slider input
  sidebarPanel(
    sliderInput("obs",
                "Number of observations:",
                min = 0,
                max = 1000,
                value = 500)
  ),

  # Show a plot of the generated distribution
  mainPanel(
    plotOutput("distPlot")
  )
))
```

`parseQueryString` *Parse a GET query string from a URL*

Description

Returns a named character vector of key-value pairs.

Usage

```
parseQueryString(str)
```

Arguments

`str`	The query string. It can have a leading `"?"` or not.

Examples

```
parseQueryString("?foo=1&bar=b%20a%20r")

## Not run:
# Example of usage within a Shiny app
shinyServer(function(input, output, clientData) {

  output$queryText <- renderText({
    query <- parseQueryString(clientData$url_search)

    # Ways of accessing the values
```

```
    if (as.numeric(query$foo) == 1) {
      # Do something
    }
    if (query[["bar"]] == "targetstring") {
      # Do something else
    }

    # Return a string with key-value pairs
    paste(names(query), query, sep = "=", collapse=", ")
  })
})

## End(Not run)
```

`plotOutput` *Create an plot output element*

Description

Render a renderPlot within an application page.

Usage

```
plotOutput(outputId, width = "100%", height = "400px",
  clickId = NULL, hoverId = NULL, hoverDelay = 300,
  hoverDelayType = c("debounce", "throttle"))
```

Arguments

`outputId`	output variable to read the plot from
`width`	Plot width. Must be a valid CSS unit (like `"100%"`, `"400px"`, `"auto"`) or a number, which will be coerced to a string and have `"px"` appended.
`height`	Plot height
`clickId`	If not `NULL`, the plot will send coordinates to the server whenever it is clicked. This information will be accessible on the `input` object using `input$clickId`. The value will be a named list or vector with `x` and `y` elements indicating the mouse position in user units.
`hoverId`	If not `NULL`, the plot will send coordinates to the server whenever the mouse pauses on the plot for more than the number of milliseconds determined by `hoverTimeout`. This information will be The value will be `NULL` if the user is not hovering, and a named list or vector with `x` and `y` elements indicating the mouse position in user units.
`hoverDelay`	The delay for hovering, in milliseconds.
`hoverDelayType`	The type of algorithm for limiting the number of hover events. Use `"throttle"` to limit the number of hover events to one every `hoverDelay` milliseconds. Use `"debounce"` to suspend events while the cursor is moving, and wait until the cursor has been at rest for `hoverDelay` milliseconds before sending an event.

Value

A plot output element that can be included in a panel

Examples

```
# Show a plot of the generated distribution
mainPanel(
  plotOutput("distPlot")
)
```

`plotPNG` *Run a plotting function and save the output as a PNG*

Description

This function returns the name of the PNG file that it generates. In essence, it calls `png()`, then `func()`, then `dev.off()`. So `func` must be a function that will generate a plot when used this way.

Usage

```
plotPNG(func, filename = tempfile(fileext = ".png"),
  width = 400, height = 400, res = 72, ...)
```

Arguments

`func`	A function that generates a plot.
`filename`	The name of the output file. Defaults to a temp file with extension `.png`.
`width`	Width in pixels.
`height`	Height in pixels.
`res`	Resolution in pixels per inch. This value is passed to `png`. Note that this affects the resolution of PNG rendering in R; it won't change the actual ppi of the browser.
`...`	Arguments to be passed through to `png`. These can be used to set the width, height, background color, etc.

Details

For output, it will try to use the following devices, in this order: quartz (via `png`), then `CairoPNG`, and finally `png`. This is in order of quality of output. Notably, plain `png` output on Linux and Windows may not antialias some point shapes, resulting in poor quality output.

In some cases, `Cairo()` provides output that looks worse than `png()`. To disable Cairo output for an app, use `options(shiny.usecairo=FALSE)`.

radioButtons *Create radio buttons*

Description

Create a set of radio buttons used to select an item from a list.

Usage

```
radioButtons(inputId, label, choices, selected = NULL)
```

Arguments

inputId	Input variable to assign the control's value to
label	Display label for the control
choices	List of values to select from (if elements of the list are named then that name rather than the value is displayed to the user)
selected	Name of initially selected item (if not specified then defaults to the first item)

Value

A set of radio buttons that can be added to a UI definition.

See Also

updateRadioButtons

Examples

```
radioButtons("dist", "Distribution type:",
             c("Normal" = "norm",
               "Uniform" = "unif",
               "Log-normal" = "lnorm",
               "Exponential" = "exp"))
```

reactive *Create a reactive expression*

Description

Wraps a normal expression to create a reactive expression. Conceptually, a reactive expression is a expression whose result will change over time.

Usage

```
reactive(x, env = parent.frame(), quoted = FALSE,
  label = NULL)

is.reactive(x)
```

Arguments

x	For `reactive`, an expression (quoted or unquoted). For `is.reactive`, an object to test.
env	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression.
quoted	Is the expression quoted? By default, this is `FALSE`. This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with 'quote()'.
label	A label for the reactive expression, useful for debugging.

Details

Reactive expressions are expressions that can read reactive values and call other reactive expressions. Whenever a reactive value changes, any reactive expressions that depended on it are marked as "invalidated" and will automatically re-execute if necessary. If a reactive expression is marked as invalidated, any other reactive expressions that recently called it are also marked as invalidated. In this way, invalidations ripple through the expressions that depend on each other.

See the Shiny tutorial for more information about reactive expressions.

Value

a function, wrapped in a S3 class "reactive"

Examples

```
values <- reactiveValues(A=1)

reactiveB <- reactive({
  values$A + 1
})

# Can use quoted expressions
reactiveC <- reactive(quote({ values$A + 2 }), quoted = TRUE)

# To store expressions for later conversion to reactive, use quote()
expr_q <- quote({ values$A + 3 })
reactiveD <- reactive(expr_q, quoted = TRUE)

# View the values from the R console with isolate()
isolate(reactiveB())
isolate(reactiveC())
isolate(reactiveD())
```

`reactiveFileReader` *Reactive file reader*

Description

Given a file path and read function, returns a reactive data source for the contents of the file.

Usage

```
reactiveFileReader(intervalMillis, session, filePath,
  readFunc, ...)
```

Arguments

`intervalMillis`	Approximate number of milliseconds to wait between checks of the file's last modified time. This can be a numeric value, or a function that returns a numeric value.
`session`	The user session to associate this file reader with, or `NULL` if none. If non-null, the reader will automatically stop when the session ends.
`filePath`	The file path to poll against and to pass to `readFunc`. This can either be a single-element character vector, or a function that returns one.
`readFunc`	The function to use to read the file; must expect the first argument to be the file path to read. The return value of this function is used as the value of the reactive file reader.
`...`	Any additional arguments to pass to `readFunc` whenever it is invoked.

Details

`reactiveFileReader` works by periodically checking the file's last modified time; if it has changed, then the file is re-read and any reactive dependents are invalidated.

The `intervalMillis`, `filePath`, and `readFunc` functions will each be executed in a reactive context; therefore, they may read reactive values and reactive expressions.

Value

A reactive expression that returns the contents of the file, and automatically invalidates when the file changes on disk (as determined by last modified time).

See Also

`reactivePoll`

Examples

```
## Not run:
# Per-session reactive file reader
shinyServer(function(input, output, session)) {
  fileData <- reactiveFileReader(1000, session, 'data.csv', read.csv)

  output$data <- renderTable({
    fileData()
  })
}

# Cross-session reactive file reader. In this example, all sessions share
# the same reader, so read.csv only gets executed once no matter how many
# user sessions are connected.
fileData <- reactiveFileReader(1000, session, 'data.csv', read.csv)
shinyServer(function(input, output, session)) {
  output$data <- renderTable({
    fileData()
  })
}

## End(Not run)
```

---

reactivePlot *Plot output (deprecated)*

Description

See renderPlot.

Usage

```
reactivePlot(func, width = "auto", height = "auto", ...)
```

Arguments

func	A function.
width	Width.
height	Height.
...	Other arguments to pass on.

`reactivePoll` *Reactive polling*

Description

Used to create a reactive data source, which works by periodically polling a non-reactive data source.

Usage

```
reactivePoll(intervalMillis, session, checkFunc,
  valueFunc)
```

Arguments

`intervalMillis`	Approximate number of milliseconds to wait between calls to `checkFunc`. This can be either a numeric value, or a function that returns a numeric value.
`session`	The user session to associate this file reader with, or `NULL` if none. If non-null, the reader will automatically stop when the session ends.
`checkFunc`	A relatively cheap function whose values over time will be tested for equality; inequality indicates that the underlying value has changed and needs to be invalidated and re-read using `valueFunc`. See Details.
`valueFunc`	A function that calculates the underlying value. See Details.

Details

`reactivePoll` works by pairing a relatively cheap "check" function with a more expensive value retrieval function. The check function will be executed periodically and should always return a consistent value until the data changes. When the check function returns a different value, then the value retrieval function will be used to re-populate the data.

Note that the check function doesn't return `TRUE` or `FALSE` to indicate whether the underlying data has changed. Rather, the check function indicates change by returning a different value from the previous time it was called.

For example, `reactivePoll` is used to implement `reactiveFileReader` by pairing a check function that simply returns the last modified timestamp of a file, and a value retrieval function that actually reads the contents of the file.

As another example, one might read a relational database table reactively by using a check function that does `SELECT MAX(timestamp) FROM table` and a value retrieval function that does `SELECT * FROM table`.

The `intervalMillis`, `checkFunc`, and `valueFunc` functions will be executed in a reactive context; therefore, they may read reactive values and reactive expressions.

Value

A reactive expression that returns the result of `valueFunc`, and invalidates when `checkFunc` changes.

See Also

reactiveFileReader

Examples

```
## Not run:
# Assume the existence of readTimestamp and readValue functions
shinyServer(function(input, output, session) {
  data <- reactivePoll(1000, session, readTimestamp, readValue)
  output$dataTable <- renderTable({
    data()
  })
})

## End(Not run)
```

---

reactivePrint *Print output (deprecated)*

Description

See renderPrint.

Usage

```
reactivePrint(func)
```

Arguments

func	A function.

---

reactiveTable *Table output (deprecated)*

Description

See renderTable.

Usage

```
reactiveTable(func, ...)
```

Arguments

func	A function.
...	Other arguments to pass on.

reactiveText *Text output (deprecated)*

Description

See renderText.

Usage

```
reactiveText(func)
```

Arguments

func	A function.

reactiveTimer *Timer*

Description

Creates a reactive timer with the given interval. A reactive timer is like a reactive value, except reactive values are triggered when they are set, while reactive timers are triggered simply by the passage of time.

Usage

```
reactiveTimer(intervalMs = 1000, session)
```

Arguments

intervalMs	How often to fire, in milliseconds
session	A session object. This is needed to cancel any scheduled invalidations after a user has ended the session. If NULL, then this invalidation will not be tied to any session, and so it will still occur.

Details

Reactive expressions and observers that want to be invalidated by the timer need to call the timer function that reactiveTimer returns, even if the current time value is not actually needed.

See invalidateLater as a safer and simpler alternative.

Value

A no-parameter function that can be called from a reactive context, in order to cause that context to be invalidated the next time the timer interval elapses. Calling the returned function also happens to yield the current time (as in Sys.time).

See Also

invalidateLater

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  # Anything that calls autoInvalidate will automatically invalidate
  # every 2 seconds.
  autoInvalidate <- reactiveTimer(2000, session)

  observe({
    # Invalidate and re-execute this reactive expression every time the
    # timer fires.
    autoInvalidate()

    # Do something each time this is invalidated.
    # The isolate() makes this observer _not_ get invalidated and re-executed
    # when input$n changes.
    print(paste("The value of input$n is", isolate(input$n)))
  })

  # Generate a new histogram each time the timer fires, but not when
  # input$n changes.
  output$plot <- renderPlot({
    autoInvalidate()
    hist(isolate(input$n))
  })
})

## End(Not run)
```

---

reactiveUI *UI output (deprecated)*

Description

See renderUI.

Usage

```
reactiveUI(func)
```

Arguments

func	A function.

`reactiveValues` *Create an object for storing reactive values*

Description

This function returns an object for storing reactive values. It is similar to a list, but with special capabilities for reactive programming. When you read a value from it, the calling reactive expression takes a reactive dependency on that value, and when you write to it, it notifies any reactive functions that depend on that value.

Usage

```
reactiveValues(...)
```

Arguments

`...`	Objects that will be added to the reactivevalues object. All of these objects must be named.

See Also

`isolate` and `is.reactivevalues`.

Examples

```
# Create the object with no values
values <- reactiveValues()

# Assign values to 'a' and 'b'
values$a <- 3
values[['b']] <- 4

## Not run:
# From within a reactive context, you can access values with:
values$a
values[['a']]

## End(Not run)

# If not in a reactive context (e.g., at the console), you can use isolate()
# to retrieve the value:
isolate(values$a)
isolate(values[['a']])

# Set values upon creation
values <- reactiveValues(a = 1, b = 2)
isolate(values$a)
```

reactiveValuesToList *Convert a reactivevalues object to a list*

Description

This function does something similar to what you might `as.list` to do. The difference is that the calling context will take dependencies on every object in the reactivevalues object. To avoid taking dependencies on all the objects, you can wrap the call with `isolate()`.

Usage

```
reactiveValuesToList(x, all.names = FALSE)
```

Arguments

`x`	A reactivevalues object.
`all.names`	If `TRUE`, include objects with a leading dot. If `FALSE` (the default) don't include those objects.

Examples

```
values <- reactiveValues(a = 1)
## Not run:
reactiveValuesToList(values)

## End(Not run)

# To get the objects without taking dependencies on them, use isolate().
# isolate() can also be used when calling from outside a reactive context (e.g.
# at the console)
isolate(reactiveValuesToList(values))
```

renderImage *Image file output*

Description

Renders a reactive image that is suitable for assigning to an `output` slot.

Usage

```
renderImage(expr, env = parent.frame(), quoted = FALSE,
  deleteFile = TRUE)
```

Arguments

expr	An expression that returns a list.
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.
deleteFile	Should the file in func()$src be deleted after it is sent to the client browser? Genrrally speaking, if the image is a temp file generated within func, then this should be TRUE; if the image is not a temp file, this should be FALSE.

Details

The expression expr must return a list containing the attributes for the img object on the client web page. For the image to display, properly, the list must have at least one entry, src, which is the path to the image file. It may also useful to have a contentType entry specifying the MIME type of the image. If one is not provided, renderImage will try to autodetect the type, based on the file extension.

Other elements such as width, height, class, and alt, can also be added to the list, and they will be used as attributes in the img object.

The corresponding HTML output tag should be div or img and have the CSS class name shiny-image-output.

See Also

For more details on how the images are generated, and how to control the output, see plotPNG.

Examples

```
## Not run:

shinyServer(function(input, output, clientData) {

  # A plot of fixed size
  output$plot1 <- renderImage({
    # A temp file to save the output. It will be deleted after renderImage
    # sends it, because deleteFile=TRUE.
    outfile <- tempfile(fileext='.png')

    # Generate a png
    png(outfile, width=400, height=400)
    hist(rnorm(input$n))
    dev.off()

    # Return a list
    list(src = outfile,
         alt = "This is alternate text")
  }, deleteFile = TRUE)

  # A dynamically-sized plot
  output$plot2 <- renderImage({
    # Read plot2's width and height. These are reactive values, so this
```

```
    # expression will re-run whenever these values change.
    width  <- clientData$output_plot2_width
    height <- clientData$output_plot2_height

    # A temp file to save the output.
    outfile <- tempfile(fileext='.png')

    png(outfile, width=width, height=height)
    hist(rnorm(input$obs))
    dev.off()

    # Return a list containing the filename
    list(src = outfile,
         width = width,
         height = height,
         alt = "This is alternate text")
  }, deleteFile = TRUE)

  # Send a pre-rendered image, and don't delete the image after sending it
  output$plot3 <- renderImage({
    # When input$n is 1, filename is ./images/image1.jpeg
    filename <- normalizePath(file.path('./images',
                              paste('image', input$n, '.jpeg', sep='')))

    # Return a list containing the filename
    list(src = filename)
  }, deleteFile = FALSE)
})


## End(Not run)
```

---

renderPlot *Plot Output*

Description

Renders a reactive plot that is suitable for assigning to an `output` slot.

Usage

```
renderPlot(expr, width = "auto", height = "auto",
  res = 72, ..., env = parent.frame(), quoted = FALSE,
  func = NULL)
```

Arguments

`expr`	An expression that generates a plot.

width	The width of the rendered plot, in pixels; or 'auto' to use the offsetWidth of the HTML element that is bound to this plot. You can also pass in a function that returns the width in pixels or 'auto'; in the body of the function you may reference reactive values and functions.
height	The height of the rendered plot, in pixels; or 'auto' to use the offsetHeight of the HTML element that is bound to this plot. You can also pass in a function that returns the width in pixels or 'auto'; in the body of the function you may reference reactive values and functions.
res	Resolution of resulting plot, in pixels per inch. This value is passed to png. Note that this affects the resolution of PNG rendering in R; it won't change the actual ppi of the browser.
...	Arguments to be passed through to png. These can be used to set the width, height, background color, etc.
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.
func	A function that generates a plot (deprecated; use expr instead).

Details

The corresponding HTML output tag should be div or img and have the CSS class name shiny-plot-output.

See Also

For more details on how the plots are generated, and how to control the output, see plotPNG.

---

renderPrint *Printable Output*

Description

Makes a reactive version of the given function that captures any printed output, and also captures its printable result (unless invisible), into a string. The resulting function is suitable for assigning to an output slot.

Usage

```
renderPrint(expr, env = parent.frame(), quoted = FALSE,
  func = NULL)
```

Arguments

expr	An expression that may print output and/or return a printable R object.
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This
func	A function that may print output and/or return a printable R object (deprecated; use expr instead).

Details

The corresponding HTML output tag can be anything (though `pre` is recommended if you need a monospace font and whitespace preserved) and should have the CSS class name `shiny-text-output`.

The result of executing `func` will be printed inside a `capture.output` call.

Note that unlike most other Shiny output functions, if the given function returns `NULL` then `NULL` will actually be visible in the output. To display nothing, make your function return `invisible()`.

See Also

`renderText` for displaying the value returned from a function, instead of the printed output.

Examples

```
isolate({

# renderPrint captures any print output, converts it to a string, and
# returns it
visFun <- renderPrint({ "foo" })
visFun()
# '[1] "foo"'

invisFun <- renderPrint({ invisible("foo") })
invisFun()
# ''

multiprintFun <- renderPrint({
  print("foo");
  "bar"
})
multiprintFun()
# '[1] "foo"\n[1] "bar"'

nullFun <- renderPrint({ NULL })
nullFun()
# 'NULL'

invisNullFun <- renderPrint({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderPrint({ 1:5 })
vecFun()
# '[1] 1 2 3 4 5'


# Contrast with renderText, which takes the value returned from the function
# and uses cat() to convert it to a string
visFun <- renderText({ "foo" })
visFun()
# 'foo'
```

```
invisFun <- renderText({ invisible("foo") })
invisFun()
# 'foo'

multiprintFun <- renderText({
  print("foo");
  "bar"
})
multiprintFun()
# 'bar'

nullFun <- renderText({ NULL })
nullFun()
# ''

invisNullFun <- renderText({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderText({ 1:5 })
vecFun()
# '1 2 3 4 5'

})
```

`renderTable` *Table Output*

Description

Creates a reactive table that is suitable for assigning to an `output` slot.

Usage

```
renderTable(expr, ..., env = parent.frame(),
  quoted = FALSE, func = NULL)
```

Arguments

`expr`	An expression that returns an R object that can be used with `xtable`.
`...`	Arguments to be passed through to `xtable` and `print.xtable`.
`env`	The environment in which to evaluate `expr`.
`quoted`	Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable.
`func`	A function that returns an R object that can be used with `xtable` (deprecated; use `expr` instead).

Details

The corresponding HTML output tag should be `div` and have the CSS class name `shiny-html-output`.

`renderText` *Text Output*

Description

Makes a reactive version of the given function that also uses `cat` to turn its result into a single-element character vector.

Usage

```
renderText(expr, env = parent.frame(), quoted = FALSE,
  func = NULL)
```

Arguments

`expr`	An expression that returns an R object that can be used as an argument to `cat`.
`env`	The environment in which to evaluate `expr`.
`quoted`	Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable.
`func`	A function that returns an R object that can be used as an argument to `cat`.(deprecated; use `expr` instead).

Details

The corresponding HTML output tag can be anything (though `pre` is recommended if you need a monospace font and whitespace preserved) and should have the CSS class name `shiny-text-output`.

The result of executing `func` will passed to `cat`, inside a `capture.output` call.

See Also

`renderPrint` for capturing the print output of a function, rather than the returned text value.

Examples

```
isolate({

# renderPrint captures any print output, converts it to a string, and
# returns it
visFun <- renderPrint({ "foo" })
visFun()
# '[1] "foo"'

invisFun <- renderPrint({ invisible("foo") })
invisFun()
```

```
# ''

multiprintFun <- renderPrint({
  print("foo");
  "bar"
})
multiprintFun()
# '[1] "foo"\n[1] "bar"'

nullFun <- renderPrint({ NULL })
nullFun()
# 'NULL'

invisNullFun <- renderPrint({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderPrint({ 1:5 })
vecFun()
# '[1] 1 2 3 4 5'


# Contrast with renderText, which takes the value returned from the function
# and uses cat() to convert it to a string
visFun <- renderText({ "foo" })
visFun()
# 'foo'

invisFun <- renderText({ invisible("foo") })
invisFun()
# 'foo'

multiprintFun <- renderText({
  print("foo");
  "bar"
})
multiprintFun()
# 'bar'

nullFun <- renderText({ NULL })
nullFun()
# ''

invisNullFun <- renderText({ invisible(NULL) })
invisNullFun()
# ''

vecFun <- renderText({ 1:5 })
vecFun()
# '1 2 3 4 5'

})
```

renderUI *UI Output*

Description

Experimental feature. Makes a reactive version of a function that generates HTML using the Shiny UI library.

Usage

```
renderUI(expr, env = parent.frame(), quoted = FALSE,
  func = NULL)
```

Arguments

expr	An expression that returns a Shiny tag object, HTML, or a list of such objects.
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.
func	A function that returns a Shiny tag object, HTML, or a list of such objects (deprecated; use expr instead).

Details

The corresponding HTML output tag should be div and have the CSS class name shiny-html-output (or use uiOutput).

See Also

conditionalPanel

Examples

```
## Not run:
  output$moreControls <- renderUI({
    list(

    )
  })

## End(Not run)
```

repeatable *Make a random number generator repeatable*

Description

Given a function that generates random data, returns a wrapped version of that function that always uses the same seed when called. The seed to use can be passed in explicitly if desired; otherwise, a random number is used.

Usage

```
repeatable(rngfunc,
  seed = runif(1, 0, .Machine$integer.max))
```

Arguments

rngfunc	The function that is affected by the R session's seed.
seed	The seed to set every time the resulting function is called.

Value

A repeatable version of the function that was passed in.

Note

When called, the returned function attempts to preserve the R session's current seed by snapshotting and restoring `.Random.seed`.

Examples

```
rnormA <- repeatable(rnorm)
rnormB <- repeatable(rnorm)
rnormA(3)  # [1]  1.8285879 -0.7468041 -0.4639111
rnormA(3)  # [1]  1.8285879 -0.7468041 -0.4639111
rnormA(5)  # [1]  1.8285879 -0.7468041 -0.4639111 -1.6510126 -1.4686924
rnormB(5)  # [1] -0.7946034  0.2568374 -0.6567597  1.2451387 -0.8375699
```

runApp *Run Shiny Application*

Description

Runs a Shiny application. This function normally does not return; interrupt R to stop the application (usually by pressing Ctrl+C or Esc).

Usage

```
runApp(appDir = getwd(), port = 8100L,
  launch.browser = getOption("shiny.launch.browser", interactive()),
  workerId = "")
```

Arguments

appDir	The directory of the application. Should contain server.R, plus, either ui.R or a www directory that contains the file index.html. Defaults to the working directory.
port	The TCP port that the application should listen on. Defaults to port 8100.
launch.browser	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.
workerId	Can generally be ignored. Exists to help some editions of Shiny Server Pro route requests to the correct process.

Examples

```
## Not run:
# Start app in the current working directory
runApp()

# Start app in a subdirectory called myapp
runApp("myapp")


# Apps can be run without a server.r and ui.r file
runApp(list(
  ui = bootstrapPage(
    numericInput('n', 'Number of obs', 100),
    plotOutput('plot')
  ),
  server = function(input, output) {
    output$plot <- renderPlot({ hist(runif(input$n)) })
  }
))

## End(Not run)
```

---

runExample *Run Shiny Example Applications*

Description

Launch Shiny example applications, and optionally, your system's web browser.

Usage

```
runExample(example = NA, port = 8100L,
  launch.browser = getOption("shiny.launch.browser", interactive()))
```

Arguments

example	The name of the example to run, or NA (the default) to list the available examples.
port	The TCP port that the application should listen on. Defaults to port 8100.
launch.browser	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.

Examples

```
## Not run:
# List all available examples
runExample()

# Run one of the examples
runExample("01_hello")

# Print the directory containing the code for all examples
system.file("examples", package="shiny")

## End(Not run)
```

---

runGist *Run a Shiny application from https://gist.github.com*

Description

Download and launch a Shiny application that is hosted on GitHub as a gist.

Usage

```
runGist(gist, port = 8100L,
  launch.browser = getOption("shiny.launch.browser", interactive()))
```

Arguments

gist	The identifier of the gist. For example, if the gist is https://gist.github.com/jcheng5/3239667, then 3239667, '3239667', and 'https://gist.github.com/jcheng5/3239667' are all valid values.
port	The TCP port that the application should listen on. Defaults to port 8100.
launch.browser	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.

Examples

```
## Not run:
runGist(3239667)
runGist("https://gist.github.com/jcheng5/3239667")

# Old URL format without username
runGist("https://gist.github.com/3239667")

## End(Not run)
```

---

runGitHub *Run a Shiny application from a GitHub repository*

Description

Download and launch a Shiny application that is hosted in a GitHub repository.

Usage

```
runGitHub(repo, username = getOption("github.user"),
  ref = "master", subdir = NULL, port = 8100,
  launch.browser = getOption("shiny.launch.browser", interactive()))
```

Arguments

repo	Name of the repository
username	GitHub username
ref	Desired git reference. Could be a commit, tag, or branch name. Defaults to "master".
subdir	A subdirectory in the repository that contains the app. By default, this function will run an app from the top level of the repo, but you can use a path such as '"inst/shinyapp".
port	The TCP port that the application should listen on. Defaults to port 8100.
launch.browser	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.

Examples

```
## Not run:
runGitHub("shiny_example", "rstudio")

# Can run an app from a subdirectory in the repo
runGitHub("shiny_example", "rstudio", subdir = "inst/shinyapp/")

## End(Not run)
```

runUrl *Run a Shiny application from a URL*

Description

Download and launch a Shiny application that is hosted at a downloadable URL. The Shiny application must be saved in a .zip, .tar, or .tar.gz file. The Shiny application files must be contained in a subdirectory in the archive. For example, the files might be `myapp/server.r` and `myapp/ui.r`.

Usage

```
runUrl(url, filetype = NULL, subdir = NULL, port = 8100,
  launch.browser = getOption("shiny.launch.browser", interactive()))
```

Arguments

`url`	URL of the application.
`filetype`	The file type (`".zip"`, `".tar"`, or `".tar.gz"`. Defaults to the file extension taken from the url.
`subdir`	A subdirectory in the repository that contains the app. By default, this function will run an app from the top level of the repo, but you can use a path such as `'"inst/shinyapp"`.
`port`	The TCP port that the application should listen on. Defaults to port 8100.
`launch.browser`	If true, the system's default web browser will be launched automatically after the app is started. Defaults to true in interactive sessions only.

Examples

```
## Not run:
runUrl('https://github.com/rstudio/shiny_example/archive/master.tar.gz')

# Can run an app from a subdirectory in the archive
runUrl("https://github.com/rstudio/shiny_example/archive/master.zip",
 subdir = "inst/shinyapp/")

## End(Not run)
```

selectInput *Create a select list input control*

Description

Create a select list that can be used to choose a single or multiple items from a list of values.

Usage

```
selectInput(inputId, label, choices, selected = NULL,
  multiple = FALSE)
```

Arguments

inputId	Input variable to assign the control's value to
label	Display label for the control
choices	List of values to select from. If elements of the list are named then that name rather than the value is displayed to the user.
selected	Name of initially selected item (or multiple names if `multiple = TRUE`). If not specified then defaults to the first item for single-select lists and no items for multiple select lists.
multiple	Is selection of multiple items allowed?

Value

A select list control that can be added to a UI definition.

See Also

updateSelectInput

Examples

```
selectInput("variable", "Variable:",
            c("Cylinders" = "cyl",
              "Transmission" = "am",
              "Gears" = "gear"))
```

---

shinyDeprecated *Print message for deprecated functions in Shiny*

Description

To disable these messages, use `options(shiny.deprecation.messages=FALSE)`.

Usage

```
shinyDeprecated(new = NULL, msg = NULL,
  old = as.character(sys.call(sys.parent()))[1L])
```

Arguments

new	Name of replacement function.
msg	Message to print. If used, this will override the default message.
old	Name of deprecated function.

shinyServer *Define Server Functionality*

Description

Defines the server-side logic of the Shiny application. This generally involves creating functions that map user inputs to various kinds of output.

Usage

```
shinyServer(func)
```

Arguments

func	The server function for this application. See the details section for more information.

Details

Call shinyServer from your application's server.R file, passing in a "server function" that provides the server-side logic of your application.

The server function will be called when each client (web browser) first loads the Shiny application's page. It must take an input and an output parameter. Any return value will be ignored. It also takes an optional session parameter, which is used when greater control is needed.

See the tutorial for more on how to write a server function.

Examples

```
## Not run:
# A very simple Shiny app that takes a message from the user
# and outputs an uppercase version of it.
shinyServer(function(input, output, session) {
  output$uppercase <- renderText({
    toupper(input$message)
  })
})

## End(Not run)
```

shinyUI *Create a Shiny UI handler*

Description

Register a UI handler by providing a UI definition (created with e.g. pageWithSidebar) and web server path (typically "/", the default value).

Usage

```
shinyUI(ui, path = "/")
```

Arguments

ui	A user-interace definition
path	The web server path to server the UI from

Value

Called for its side-effect of registering a UI handler

Examples

```
el <- div(HTML("I like <u>turtles</u>"))
cat(as.character(el))
```

showReactLog *Reactive Log Visualizer*

Description

Provides an interactive browser-based tool for visualizing reactive dependencies and execution in your application.

Usage

```
showReactLog()
```

Details

To use the reactive log visualizer, start with a fresh R session and run the command `options(shiny.reactlog=TRUE)`; then launch your application in the usual way (e.g. using `runApp`). At any time you can hit Ctrl+F3 (or for Mac users, Command+F3) in your web browser to launch the reactive log visualization.

The reactive log visualization only includes reactive activity up until the time the report was loaded. If you want to see more recent activity, refresh the browser.

Note that Shiny does not distinguish between reactive dependencies that "belong" to one Shiny user session versus another, so the visualization will include all reactive activity that has taken place in the process, not just for a particular application or session.

As an alternative to pressing Ctrl/Command+F3–for example, if you are using reactives outside of the context of a Shiny application–you can run the `showReactLog` function, which will generate the reactive log visualization as a static HTML file and launch it in your default browser. In this case, refreshing your browser will not load new activity into the report; you will need to call `showReactLog()` explicitly.

For security and performance reasons, do not enable `shiny.reactlog` in production environments. When the option is enabled, it's possible for any user of your app to see at least some of the source code of your reactive expressions and observers.

`sidebarPanel` *Create a sidebar panel*

Description

Create a sidebar panel containing input controls that can in turn be passed to pageWithSidebar.

Usage

```
sidebarPanel(...)
```

Arguments

`...`	UI elements to include on the sidebar

Value

A sidebar that can be passed to pageWithSidebar

Examples

```
# Sidebar with controls to select a dataset and specify
# the number of observations to view
sidebarPanel(
  selectInput("dataset", "Choose a dataset:",
              choices = c("rock", "pressure", "cars")),

  numericInput("obs", "Observations:", 10)
)
```

singleton *Include Content Only Once*

Description

Use `singleton` to wrap contents (tag, text, HTML, or lists) that should be included in the generated document only once, yet may appear in the document-generating code more than once. Only the first appearance of the content (in document order) will be used. Useful for custom components that have JavaScript files or stylesheets.

Usage

```
singleton(x)
```

Arguments

`x`	A `tag`, text, `HTML`, or list.

sliderInput *Slider Input Widget*

Description

Constructs a slider widget to select a numeric value from a range.

Usage

```
sliderInput(inputId, label, min, max, value, step = NULL,
  round = FALSE, format = "#,##0.#####", locale = "us",
  ticks = TRUE, animate = FALSE)
```

Arguments

`inputId`	Specifies the `input` slot that will be used to access the value.
`label`	A descriptive label to be displayed with the widget.
`min`	The minimum value (inclusive) that can be selected.
`max`	The maximum value (inclusive) that can be selected.
`value`	The initial value of the slider. A numeric vector of length one will create a regular slider; a numeric vector of length two will create a double-ended range slider.. A warning will be issued if the value doesn't fit between `min` and `max`.
`step`	Specifies the interval between each selectable value on the slider (`NULL` means no restriction).

round	TRUE to round all values to the nearest integer; FALSE if no rounding is desired; or an integer to round to that number of digits (for example, 1 will round to the nearest 10, and -2 will round to the nearest .01). Any rounding will be applied after snapping to the nearest step.
format	Customize format values in slider labels. See http://archive.plugins.jquery.com/project/numberformatter for syntax details.
locale	The locale to be used when applying format. See details.
ticks	FALSE to hide tick marks, TRUE to show them according to some simple heuristics.
animate	TRUE to show simple animation controls with default settings; FALSE not to; or a custom settings list, such as those created using animationOptions.

Details

Valid values for locale are:

Arab Emirates	"ae"
Australia	"au"
Austria	"at"
Brazil	"br"
Canada	"ca"
China	"cn"
Czech	"cz"
Denmark	"dk"
Egypt	"eg"
Finland	"fi"
France	"fr"
Germany	"de"
Greece	"gr"
Great Britain	"gb"
Hong Kong	"hk"
India	"in"
Israel	"il"
Japan	"jp"
Russia	"ru"
South Korea	"kr"
Spain	"es"
Sweden	"se"
Switzerland	"ch"
Taiwan	"tw"
Thailand	"th"
United States	"us"
Vietnam	"vn"

See Also

updateSliderInput

stopApp *Stop the currently running Shiny app*

Description

Stops the currently running Shiny app, returning control to the caller of runApp.

Usage

```
stopApp(returnValue = NULL)
```

Arguments

returnValue	The value that should be returned from runApp.

submitButton *Create a submit button*

Description

Create a submit button for an input form. Forms that include a submit button do not automatically update their outputs when inputs change, rather they wait until the user explicitly clicks the submit button.

Usage

```
submitButton(text = "Apply Changes")
```

Arguments

text	Button caption

Value

A submit button that can be added to a UI definition.

Examples

```
submitButton("Update View")
```

tableOutput *Create a table output element*

Description

Render a renderTable within an application page.

Usage

```
tableOutput(outputId)
```

Arguments

outputId	output variable to read the table from

Value

A table output element that can be included in a panel

Examples

```
mainPanel(
  tableOutput("view")
)
```

tabPanel *Create a tab panel*

Description

Create a tab panel that can be included within a tabsetPanel.

Usage

```
tabPanel(title, ..., value = NULL)
```

Arguments

title	Display title for tab
...	UI elements to include within the tab
value	The value that should be sent when tabsetPanel reports that this tab is selected. If omitted and tabsetPanel has an id, then the title will be used.

Value

A tab that can be passed to tabsetPanel

See Also

tabsetPanel

Examples

```
# Show a tabset that includes a plot, summary, and
# table view of the generated distribution
mainPanel(
  tabsetPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
)
```

---

tabsetPanel *Create a tabset panel*

Description

Create a tabset that contains tabPanel elements. Tabsets are useful for dividing output into multiple independently viewable sections.

Usage

```
tabsetPanel(..., id = NULL, selected = NULL)
```

Arguments

...	tabPanel elements to include in the tabset
id	If provided, you can use input$id in your server logic to determine which of the current tabs is active. The value will correspond to the value argument that is passed to tabPanel.
selected	The value (or, if none was supplied, the title) of the tab that should be selected by default. If NULL, the first tab will be selected.

Value

A tabset that can be passed to mainPanel

See Also

tabPanel, updateTabsetPanel

Examples

```
# Show a tabset that includes a plot, summary, and
# table view of the generated distribution
mainPanel(
  tabsetPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
)
```

---

tag *HTML Tag Object*

Description

tag creates an HTML tag definition. Note that all of the valid HTML5 tags are already defined in the tags environment so these functions should only be used to generate additional tags. tagAppendChild and tagList are for supporting package authors who wish to create their own sets of tags; see the contents of bootstrap.R for examples.

```
tag(_tag_name, varArgs)

tagAppendChild(tag, child)

tagAppendChildren(tag, child1, child2)

tagAppendChildren(tag, list = list(child1, child2))

tagSetChildren(tag, child1, child2)

tagSetChildren(tag, list = list(child1, child2))

tagList(...)
```

Arguments

_tag_name	HTML tag name
varArgs	List of attributes and children of the element. Named list items become attributes, and unnamed list items become children. Valid children are tags, single-character character vectors (which become text nodes), and raw HTML (see HTML). You can also pass lists that contain tags, text nodes, and HTML.
tag	A tag to append child elements to.
child	A child element to append to a parent tag.
...	Unnamed items that comprise this list of tags.
list	An optional list of elements. Can be used with or instead of the ... items.

Value

An HTML tag object that can be rendered as HTML using as.character.

Examples

```
tagList(tags$h1("Title"),
        tags$h2("Header text"),
        tags$p("Text here"))

# Can also convert a regular list to a tagList (internal data structure isn't
# exactly the same, but when rendered to HTML, the output is the same).
x <- list(tags$h1("Title"),
          tags$h2("Header text"),
          tags$p("Text here"))
tagList(x)
```

---

textInput *Create a text input control*

Description

Create an input control for entry of unstructured text values

Usage

```
textInput(inputId, label, value = "")
```

Arguments

inputId	Input variable to assign the control's value to
label	Display label for the control
value	Initial value

Value

A text input control that can be added to a UI definition.

See Also

updateTextInput

Examples

```
textInput("caption", "Caption:", "Data Summary")
```

textOutput *Create a text output element*

Description

Render a reactive output variable as text within an application page. The text will be included within an HTML `div` tag.

Usage

```
textOutput(outputId)
```

Arguments

`outputId`	output variable to read the value from

Details

Text is HTML-escaped prior to rendering. This element is often used to display renderText output variables.

Value

A text output element that can be included in a panel

Examples

```
h3(textOutput("caption"))
```

updateCheckboxGroupInput *Change the value of a checkbox group input on the client*

Description

Change the value of a checkbox group input on the client

Usage

```
updateCheckboxGroupInput(session, inputId, label = NULL,
  choices = NULL, selected = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the input object.
label	The label to set for the input object.
choices	A named vector or named list of options. For each item, the name will be used as the label, and the value will be used as the value.
selected	A vector or list of options which will be selected.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, numericInput() and updateNumericInput() take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

checkboxGroupInput

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    # Create a list of new options, where the name of the items is something
    # like 'option label x 1', and the values are 'option-x-1'.
    cb_options <- list()
    cb_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)
    cb_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)

    # Change values for input$inCheckboxGroup
    updateCheckboxGroupInput(session, "inCheckboxGroup", choices = cb_options)

    # Can also set the label and select items
    updateCheckboxGroupInput(session, "inCheckboxGroup2",
      label = paste("checkboxgroup label", x),
      choices = cb_options,
      selected = sprintf("option label %d 2", x)
    )
  })
})
```

```
## End(Not run)
```

`updateCheckboxInput` *Change the value of a checkbox input on the client*

Description

Change the value of a checkbox input on the client

Usage

```
updateCheckboxInput(session, inputId, label = NULL,
  value = NULL)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`inputId`	The id of the input object.
`label`	The label to set for the input object.
`value`	The value to set for the input object.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

`checkboxInput`

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # TRUE if input$controller is even, FALSE otherwise.
    x_even <- input$controller %% 2 == 0

    updateCheckboxInput(session, "inCheckbox", value = x_even)
  })
```

```
})

## End(Not run)
```

`updateDateInput` *Change the value of a date input on the client*

Description

Change the value of a date input on the client

Usage

```
updateDateInput(session, inputId, label = NULL,
  value = NULL, min = NULL, max = NULL)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`inputId`	The id of the input object.
`label`	The label to set for the input object.
`value`	The desired date value. Either a Date object, or a string in `yyyy-mm-dd` format.
`min`	The minimum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`max`	The maximum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

`dateInput`

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    updateDateInput(session, "inDate",
      label = paste("Date label", x),
      value = paste("2013-04-", x, sep=""),
      min   = paste("2013-04-", x-1, sep=""),
      max   = paste("2013-04-", x+1, sep="")
    )
  })
})

## End(Not run)
```

`updateDateRangeInput` *Change the start and end values of a date range input on the client*

Description

Change the start and end values of a date range input on the client

Usage

```
updateDateRangeInput(session, inputId, label = NULL,
  start = NULL, end = NULL, min = NULL, max = NULL)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`inputId`	The id of the input object.
`label`	The label to set for the input object.
`start`	The start date. Either a Date object, or a string in `yyyy-mm-dd` format.
`end`	The end date. Either a Date object, or a string in `yyyy-mm-dd` format.
`min`	The minimum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`max`	The maximum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

`dateRangeInput`

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    updateDateRangeInput(session, "inDateRange",
      label = paste("Date range label", x),
      start = paste("2013-01-", x, sep=""))
      end = paste("2013-12-", x, sep=""))
  })
})

## End(Not run)
```

---

updateNumericInput *Change the value of a number input on the client*

Description

Change the value of a number input on the client

Usage

```
updateNumericInput(session, inputId, label = NULL,
  value = NULL, min = NULL, max = NULL, step = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the input object.
label	The label to set for the input object.
value	The value to set for the input object.
min	Minimum value.
max	Maximum value.
step	Step size.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, numericInput() and updateNumericInput() take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

numericInput

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    updateNumericInput(session, "inNumber", value = x)

    updateNumericInput(session, "inNumber2",
      label = paste("Number label ", x),
      value = x, min = x-10, max = x+10, step = 5)
  })
})

## End(Not run)
```

updateRadioButtons *Change the value of a radio input on the client*

Description

Change the value of a radio input on the client

Usage

```
updateRadioButtons(session, inputId, label = NULL,
  choices = NULL, selected = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the input object.
label	The label to set for the input object.
choices	A named vector or named list of options. For each item, the name will be used as the label, and the value will be used as the value.
selected	A vector or list of options which will be selected.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, numericInput() and updateNumericInput() take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

radioButtons

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    r_options <- list()
```

```
    r_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)
    r_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)

    # Change values for input$inRadio
    updateRadioButtons(session, "inRadio", choices = r_options)

    # Can also set the label and select an item
    updateRadioButtons(session, "inRadio2",
      label = paste("Radio label", x),
      choices = r_options,
      selected = sprintf("option label %d 2", x)
    )
  })
})

## End(Not run)
```

`updateSelectInput` *Change the value of a select input on the client*

Description

Change the value of a select input on the client

Usage

```
updateSelectInput(session, inputId, label = NULL,
  choices = NULL, selected = NULL)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`inputId`	The id of the input object.
`label`	The label to set for the input object.
`choices`	A named vector or named list of options. For each item, the name will be used as the label, and the value will be used as the value.
`selected`	A vector or list of options which will be selected.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, `numericInput()` and `updateNumericInput()` take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

selectInput

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    # Create a list of new options, where the name of the items is something
    # like 'option label x 1', and the values are 'option-x-1'.
    s_options <- list()
    s_options[[sprintf("option label %d 1", x)]] <- sprintf("option-%d-1", x)
    s_options[[sprintf("option label %d 2", x)]] <- sprintf("option-%d-2", x)

    # Change values for input$inSelect
    updateSelectInput(session, "inSelect", choices = s_options)

    # Can also set the label and select an item (or more than one if it's a
    # multi-select)
    updateSelectInput(session, "inSelect2",
      label = paste("Select label", x),
      choices = s_options,
      selected = sprintf("option label %d 2", x)
    )
  })
})

## End(Not run)
```

---

updateSliderInput *Change the value of a slider input on the client*

Description

Change the value of a slider input on the client

Usage

```
updateSliderInput(session, inputId, label = NULL,
  value = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the input object.
label	The label to set for the input object.
value	The value to set for the input object.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, numericInput() and updateNumericInput() take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

sliderInput

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    # Similar to number and text. only label and value can be set for slider
    updateSliderInput(session, "inSlider",
      label = paste("Slider label", x),
      value = x)

    # For sliders that pick out a range, pass in a vector of 2 values.
    updateSliderInput(session, "inSlider2", value = c(x-1, x+1))

    # An NA means to not change that value (the low or high one)
    updateSliderInput(session, "inSlider3", value = c(NA, x+2))
  })
})

## End(Not run)
```

updateTabsetPanel *Change the selected tab on the client*

Description

Change the selected tab on the client

Usage

```
updateTabsetPanel(session, inputId, selected = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the tabset panel object.
selected	The name of the tab to make active.

See Also

tabsetPanel

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # TRUE if input$controller is even, FALSE otherwise.
    x_even <- input$controller %% 2 == 0

    # Change the selected tab.
    # Note that the tabsetPanel must have been created with an 'id' argument
    if (x_even) {
      updateTabsetPanel(session, "inTabset", selected = "panel2")
    } else {
      updateTabsetPanel(session, "inTabset", selected = "panel1")
    }
  })
})

## End(Not run)
```

updateTextInput *Change the value of a text input on the client*

Description

Change the value of a text input on the client

Usage

```
updateTextInput(session, inputId, label = NULL,
  value = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
inputId	The id of the input object.
label	The label to set for the input object.
value	The value to set for the input object.

Details

The input updater functions send a message to the client, telling it to change the settings of an input object. The messages are collected and sent after all the observers (including outputs) have finished running.

The syntax of these functions is similar to the functions that created the inputs in the first place. For example, numericInput() and updateNumericInput() take a similar set of arguments.

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.

See Also

textInput

Examples

```
## Not run:
shinyServer(function(input, output, session) {

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller

    # This will change the value of input$inText, based on x
    updateTextInput(session, "inText", value = paste("New text", x))

    # Can also set the label, this time for input$inText2
```

```
    updateTextInput(session, "inText2",
      label = paste("New label", x),
      value = paste("New text", x))
  })
})

## End(Not run)
```

---

validateCssUnit *Validate proper CSS formatting of a unit*

Description

Validate proper CSS formatting of a unit

Usage

```
validateCssUnit(x)
```

Arguments

x	The unit to validate. Will be treated as a number of pixels if a unit is not specified.

Value

A properly formatted CSS unit of length, if possible. Otherwise, will throw an error.

Examples

```
validateCssUnit("10%")
validateCssUnit(400)  #treated as '400px'
```

---

verbatimTextOutput *Create a verbatim text output element*

Description

Render a reactive output variable as verbatim text within an application page. The text will be included within an HTML `pre` tag.

Usage

```
verbatimTextOutput(outputId)
```

Arguments

outputId	output variable to read the value from

Details

Text is HTML-escaped prior to rendering. This element is often used with the renderPrint function to preserve fixed-width formatting of printed objects.

Value

A verbatim text output element that can be included in a panel

Examples

```
mainPanel(
  h4("Summary"),
  verbatimTextOutput("summary"),

  h4("Observations"),
  tableOutput("view")
)
```

---

wellPanel *Create a well panel*

Description

Creates a panel with a slightly inset border and grey background. Equivalent to Twitter Bootstrap's `well` CSS class.

Usage

```
wellPanel(...)
```

Arguments

...	UI elements to include inside the panel.

Value

The newly created panel.

withTags *Evaluate an expression using the* tags

Description

This function makes it simpler to write HTML-generating code. Instead of needing to specify tags each time a tag function is used, as in tags$div() and tags$p(), code inside withTags is evaluated with tags searched first, so you can simply use div() and p().

Usage

```
withTags(code)
```

Arguments

code	A set of tags.

Details

If your code uses an object which happens to have the same name as an HTML tag function, such as source() or summary(), it will call the tag function. To call the intended (non-tags function), specify the namespace, as in base::source() or base::summary().

Examples

```
# Using tags$ each time
tags$div(class = "myclass",
  tags$h3("header"),
  tags$p("text")
)

# Equivalent to above, but using withTags
withTags(
  div(class = "myclass",
    h3("header"),
    p("text")
  )
)
```

Index