# Package 'shiny'

March 13, 2020

**Type** Package

**Title** Web Application Framework for R

**Version** 1.4.0.2

**Description** Makes it incredibly easy to build interactive web
applications with R. Automatic ``reactive'' binding between inputs and
outputs and extensive prebuilt widgets make it possible to build
beautiful, responsive, and powerful applications with minimal effort.

**License** GPL-3 | file LICENSE

**Depends** R (>= 3.0.2), methods

**Imports** utils, grDevices, httpuv (>= 1.5.2), mime (>= 0.3), jsonlite
(>= 0.9.16), xtable, digest, htmltools (>= 0.4.0), R6 (>= 2.0),
sourcetools, later (>= 1.0.0), promises (>= 1.1.0), tools,
crayon, rlang (>= 0.4.0), fastmap (>= 1.0.0)

**Suggests** datasets, Cairo (>= 1.5-5), testthat (>= 2.1.1), knitr (>=
1.6), markdown, rmarkdown, ggplot2, reactlog (>= 1.0.0),
magrittr, yaml

**URL** <http://shiny.rstudio.com>

**BugReports** <https://github.com/rstudio/shiny/issues>

**Collate** 'app.R' 'bookmark-state-local.R' 'stack.R' 'bookmark-state.R'
'bootstrap-deprecated.R' 'bootstrap-layout.R' 'globals.R'
'conditions.R' 'map.R' 'utils.R' 'bootstrap.R'
'cache-context.R' 'cache-disk.R' 'cache-memory.R'
'cache-utils.R' 'diagnose.R' 'fileupload.R' 'font-awesome.R'
'graph.R' 'reactives.R' 'reactive-domains.R' 'history.R'
'hooks.R' 'html-deps.R' 'htmltools.R' 'image-interact-opts.R'
'image-interact.R' 'imageutils.R' 'input-action.R'
'input-checkbox.R' 'input-checkboxgroup.R' 'input-date.R'
'input-daterange.R' 'input-file.R' 'input-numeric.R'
'input-password.R' 'input-radiobuttons.R' 'input-select.R'
'input-slider.R' 'input-submit.R' 'input-text.R'
'input-textarea.R' 'input-utils.R' 'insert-tab.R' 'insert-ui.R'
'jqueryui.R' 'middleware-shiny.R' 'middleware.R' 'modal.R'

'modules.R' 'notifications.R' 'priorityqueue.R' 'progress.R'
'react.R' 'render-cached-plot.R' 'render-plot.R'
'render-table.R' 'run-url.R' 'serializers.R'
'server-input-handlers.R' 'server.R' 'shiny-options.R'
'shiny.R' 'shinyui.R' 'shinywrappers.R' 'showcase.R'
'snapshot.R' 'tar.R' 'test-export.R' 'timer.R' 'update-input.R'

**RoxygenNote** 6.1.1

**Encoding** UTF-8

**NeedsCompilation** no

**Author** Winston Chang [aut, cre],
Joe Cheng [aut],
JJ Allaire [aut],
Yihui Xie [aut],
Jonathan McPherson [aut],
RStudio [cph],
jQuery Foundation [cph] (jQuery library and jQuery UI library),
jQuery contributors [ctb, cph] (jQuery library; authors listed in
inst/www/shared/jquery-AUTHORS.txt),
jQuery UI contributors [ctb, cph] (jQuery UI library; authors listed in
inst/www/shared/jqueryui/AUTHORS.txt),
Mark Otto [ctb] (Bootstrap library),
Jacob Thornton [ctb] (Bootstrap library),
Bootstrap contributors [ctb] (Bootstrap library),
Twitter, Inc [cph] (Bootstrap library),
Alexander Farkas [ctb, cph] (html5shiv library),
Scott Jehl [ctb, cph] (Respond.js library),
Stefan Petre [ctb, cph] (Bootstrap-datepicker library),
Andrew Rowls [ctb, cph] (Bootstrap-datepicker library),
Dave Gandy [ctb, cph] (Font-Awesome font),
Brian Reavis [ctb, cph] (selectize.js library),
Kristopher Michael Kowal [ctb, cph] (es5-shim library),
es5-shim contributors [ctb, cph] (es5-shim library),
Denis Ineshin [ctb, cph] (ion.rangeSlider library),
Sami Samhuri [ctb, cph] (Javascript strftime library),
SpryMedia Limited [ctb, cph] (DataTables library),
John Fraser [ctb, cph] (showdown.js library),
John Gruber [ctb, cph] (showdown.js library),
Ivan Sagalaev [ctb, cph] (highlight.js library),
R Core Team [ctb, cph] (tar implementation from R)

**Maintainer** Winston Chang <winston@rstudio.com>

**Repository** CRAN

**Date/Publication** 2020-03-13 10:00:02 UTC

# 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 prebuilt widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.

### Details

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

### See Also

[shiny-options](#) for documentation about global options.

---

absolutePanel                      *Panel with absolute positioning*

---

### Description

Creates a panel whose contents are absolutely positioned.

## Usage

```
absolutePanel(..., top = NULL, left = NULL, right = NULL,
  bottom = NULL, width = NULL, height = NULL, draggable = FALSE,
  fixed = FALSE, cursor = c("auto", "move", "default", "inherit"))

fixedPanel(..., top = NULL, left = NULL, right = NULL,
  bottom = NULL, width = NULL, height = NULL, draggable = FALSE,
  cursor = c("auto", "move", "default", "inherit"))
```

## Arguments

| | |
|---|---|
| `...` | Attributes (named arguments) or children (unnamed arguments) that should be included in the panel. |
| `top` | Distance between the top of the panel, and the top of the page or parent container. |
| `left` | Distance between the left side of the panel, and the left of the page or parent container. |
| `right` | Distance between the right side of the panel, and the right of the page or parent container. |
| `bottom` | Distance between the bottom of the panel, and the bottom of the page or parent container. |
| `width` | Width of the panel. |
| `height` | Height of the panel. |
| `draggable` | If TRUE, allows the user to move the panel by clicking and dragging. |
| `fixed` | Positions the panel relative to the browser window and prevents it from being scrolled with the rest of the page. |
| `cursor` | The type of cursor that should appear when the user mouses over the panel. Use `"move"` for a north-east-south-west icon, `"default"` for the usual cursor arrow, or `"inherit"` for the usual cursor behavior (including changing to an I-beam when the cursor is over text). The default is `"auto"`, which is equivalent to `ifelse(draggable,"move","inherit")`. |

## Details

The `absolutePanel` function creates a `<div>` tag whose CSS position is set to `absolute` (or `fixed` if `fixed = TRUE`). The way absolute positioning works in HTML is that absolute coordinates are specified relative to its nearest parent element whose position is not set to `static` (which is the default), and if no such parent is found, then relative to the page borders. If you're not sure what that means, just keep in mind that you may get strange results if you use `absolutePanel` from inside of certain types of panels.

The `fixedPanel` function is the same as `absolutePanel` with `fixed = TRUE`.

The position (`top`, `left`, `right`, `bottom`) and size (`width`, `height`) parameters are all optional, but you should specify exactly two of `top`, `bottom`, and `height` and exactly two of `left`, `right`, and `width` for predictable results.

Like most other distance parameters in Shiny, the position and size parameters take a number (interpreted as pixels) or a valid CSS size string, such as ″100px″ (100 pixels) or ″25%″.

For arcane HTML reasons, to have the panel fill the page or parent you should specify 0 for top, left, right, and bottom rather than the more obvious width = ″100%″ and height = ″100%″.

### Value

An HTML element or list of elements.

---

actionButton                    *Action button/link*

---

### Description

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

### Usage

```
actionButton(inputId, label, icon = NULL, width = NULL, ...)

actionLink(inputId, label, icon = NULL, ...)
```

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | The contents of the button or link–usually a text label, but you could also use any other HTML, like an image. |
| icon | An optional [icon()](#) to appear on the button. |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](#). |
| ... | Named attributes to be applied to the button or link. |

### Server value

An integer of class ″shinyActionButtonValue″. This class differs from ordinary integers in that a value of 0 is considered "falsy". This implies two things:

- Event handlers (e.g., [observeEvent()](#), [eventReactive()](#)) won't execute on initial load.
- Input validation (e.g., [req()](#), [need()](#)) will fail on initial load.

### See Also

[observeEvent()](#) and [eventReactive()](#)

Other input elements: [checkboxGroupInput](#), [checkboxInput](#), [dateInput](#), [dateRangeInput](#), [fileInput](#), [numericInput](#), [passwordInput](#), [radioButtons](#), [selectInput](#), [sliderInput](#), [submitButton](#), [textAreaInput](#), [textInput](#), [varSelectInput](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("obs", "Number of observations", 0, 1000, 500),
  actionButton("goButton", "Go!"),
  plotOutput("distPlot")
)

server <- function(input, output) {
  output$distPlot <- renderPlot({
    # Take a dependency on input$goButton. This will run once initially,
    # because the value changes from NULL to 0.
    input$goButton

    # Use isolate() to avoid dependency on input$obs
    dist <- isolate(rnorm(input$obs))
    hist(dist)
  })
}

shinyApp(ui, server)

}
```

---

addResourcePath *Resource Publishing*

---

## Description

Add, remove, or list 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)

resourcePaths()

removeResourcePath(prefix)
```

## Arguments

prefix          The URL prefix (without slashes). Valid characters are a-z, A-Z, 0-9, hyphen, period, and underscore. 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**

Shiny provides two ways of serving static files (i.e., resources):

1. Static files under the www/ directory are automatically made available under a request path that begins with /.

2. addResourcePath() makes static files in a directoryPath available under a request path that begins with prefix.

The second approach is primarily intended for package authors to make supporting JavaScript/CSS files available to their components.

Tools for managing static resources published by Shiny's web server:

- addResourcePath() adds a directory of static resources.

- resourcePaths() lists the currently active resource mappings.

- removeResourcePath() removes a directory of static resources.

**See Also**

[singleton()](#)

**Examples**

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

# make sure all resources are removed
lapply(names(resourcePaths()), removeResourcePath)
```

---

bookmarkButton                    *Create a button for bookmarking/sharing*

---

**Description**

A bookmarkButton is a [actionButton()](#) with a default label that consists of a link icon and the text "Bookmark...". It is meant to be used for bookmarking state.

**Usage**

```
bookmarkButton(label = "Bookmark...", icon = shiny::icon("link", lib =
  "glyphicon"),
  title = "Bookmark this application's state and get a URL for sharing.",
  ..., id = "._bookmark_")
```

**Arguments**

| | |
|---|---|
| `label` | The contents of the button or link–usually a text label, but you could also use any other HTML, like an image. |
| `icon` | An optional `icon()` to appear on the button. |
| `title` | A tooltip that is shown when the mouse cursor hovers over the button. |
| `...` | Named attributes to be applied to the button or link. |
| `id` | An ID for the bookmark button. The only time it is necessary to set the ID unless you have more than one bookmark button in your application. If you specify an input ID, it should be excluded from bookmarking with `setBookmarkExclude()`, and you must create an observer that does the bookmarking when the button is pressed. See the examples below. |

**See Also**

`enableBookmarking()` for more examples.

**Examples**

```
## Only run these examples in interactive sessions
if (interactive()) {

# This example shows how to use multiple bookmark buttons. If you only need
# a single bookmark button, see examples in ?enableBookmarking.
ui <- function(request) {
  fluidPage(
    tabsetPanel(id = "tabs",
      tabPanel("One",
        checkboxInput("chk1", "Checkbox 1"),
        bookmarkButton(id = "bookmark1")
      ),
      tabPanel("Two",
        checkboxInput("chk2", "Checkbox 2"),
        bookmarkButton(id = "bookmark2")
      )
    )
  )
}
server <- function(input, output, session) {
  # Need to exclude the buttons from themselves being bookmarked
  setBookmarkExclude(c("bookmark1", "bookmark2"))

  # Trigger bookmarking with either button
  observeEvent(input$bookmark1, {
    session$doBookmark()
  })
  observeEvent(input$bookmark2, {
    session$doBookmark()
  })
}
enableBookmarking(store = "url")
```

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

bootstrapLib          *Bootstrap libraries*

### Description

This function returns a set of web dependencies necessary for using Bootstrap components in a web page.

### Usage

```
bootstrapLib(theme = NULL)
```

### Arguments

theme          Alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css)

### Details

It isn't necessary to call this function if you use bootstrapPage() or others which use bootstrapPage, such basicPage(), fluidPage(), fillPage(), pageWithSidebar(), and navbarPage(), because they already include the Bootstrap web dependencies.

bootstrapPage          *Create a Bootstrap page*

### Description

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

### Usage

```
bootstrapPage(..., title = NULL, responsive = NULL, theme = NULL)

basicPage(...)
```

### Arguments

| ... | The contents of the document body. |
| --- | --- |
| title | The browser window title (defaults to the host URL of the page) |
| responsive | This option is deprecated; it is no longer optional with Bootstrap 3. |
| theme | Alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css) |

**Details**

This function is primarily intended for users who are proficient in HTML/CSS, and know how to lay out pages in Bootstrap. Most applications should use `fluidPage()` along with layout functions like `fluidRow()` and `sidebarLayout()`.

**Value**

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

**Note**

The `basicPage` function is deprecated, you should use the `fluidPage()` function instead.

**See Also**

`fluidPage()`, `fixedPage()`

---

brushedPoints                    *Find rows of data that are selected by a brush*

---

**Description**

This function returns rows from a data frame which are under a brush used with `plotOutput()`.

**Usage**

```
brushedPoints(df, brush, xvar = NULL, yvar = NULL, panelvar1 = NULL,
  panelvar2 = NULL, allRows = FALSE)
```

**Arguments**

| | |
|---|---|
| df | A data frame from which to select rows. |
| brush | The data from a brush, such as input$plot_brush. |
| xvar, yvar | A string with the name of the variable on the x or y axis. This must also be the name of a column in df. If absent, then this function will try to infer the variable from the brush (only works for ggplot2). |
| panelvar1, panelvar2 | |
| | Each of these is a string with the name of a panel variable. For example, if with ggplot2, you facet on a variable called cyl, then you can use "cyl" here. However, specifying the panel variable should not be necessary with ggplot2; Shiny should be able to auto-detect the panel variable. |
| allRows | If FALSE (the default) return a data frame containing the selected rows. If TRUE, the input data frame will have a new column, selected_, which indicates whether the row was inside the brush (TRUE) or outside the brush (FALSE). |

**Details**

It is also possible for this function to return all rows from the input data frame, but with an additional column selected_, which indicates which rows of the input data frame are selected by the brush (TRUE for selected, FALSE for not-selected). This is enabled by setting allRows=TRUE option.

The xvar, yvar, panelvar1, and panelvar2 arguments specify which columns in the data correspond to the x variable, y variable, and panel variables of the plot. For example, if your plot is plot(x=cars$speed,y=cars$dist), and your brush is named "cars_brush", then you would use brushedPoints(cars,input$cars_brush,"speed","dist").

For plots created with ggplot2, it should not be necessary to specify the column names; that information will already be contained in the brush, provided that variables are in the original data, and not computed. For example, with ggplot(cars,aes(x=speed,y=dist)) + geom_point(), you could use brushedPoints(cars,input$cars_brush). If, however, you use a computed column, like ggplot(cars,aes(x=speed/2,y=dist)) + geom_point(), then it will not be able to automatically extract column names and filter on them. If you want to use this function to filter data, it is recommended that you not use computed columns; instead, modify the data first, and then make the plot with "raw" columns in the modified data.

If a specified x or y column is a factor, then it will be coerced to an integer vector. If it is a character vector, then it will be coerced to a factor and then integer vector. This means that the brush will be considered to cover a given character/factor value when it covers the center value.

If the brush is operating in just the x or y directions (e.g., with brushOpts(direction = "x"), then this function will filter out points using just the x or y variable, whichever is appropriate.

**See Also**

[plotOutput()](#) for example usage.

---

brushOpts                    *Create an object representing brushing options*

---

**Description**

This generates an object representing brushing options, to be passed as the brush argument of [imageOutput()](#) or [plotOutput()](#).

**Usage**

```
brushOpts(id = NULL, fill = "#9cf", stroke = "#036",
  opacity = 0.25, delay = 300, delayType = c("debounce", "throttle"),
  clip = TRUE, direction = c("xy", "x", "y"), resetOnNew = FALSE)
```

**Arguments**

id                 Input value name. For example, if the value is "plot_brush", then the coordinates will be available as input$plot_brush. Multiple imageOutput/plotOutput calls may share the same id value; brushing one image or plot will cause any other brushes with the same id to disappear.

| | |
|---|---|
| `fill` | Fill color of the brush. |
| `stroke` | Outline color of the brush. |
| `opacity` | Opacity of the brush |
| `delay` | How long to delay (in milliseconds) when debouncing or throttling, before sending the brush data to the server. |
| `delayType` | The type of algorithm for limiting the number of brush events. Use `"throttle"` to limit the number of brush events to one every `delay` milliseconds. Use `"debounce"` to suspend events while the cursor is moving, and wait until the cursor has been at rest for `delay` milliseconds before sending an event. |
| `clip` | Should the brush area be clipped to the plotting area? If FALSE, then the user will be able to brush outside the plotting area, as long as it is still inside the image. |
| `direction` | The direction for brushing. If `"xy"`, the brush can be drawn and moved in both x and y directions. If `"x"`, or `"y"`, the brush wil work horizontally or vertically. |
| `resetOnNew` | When a new image is sent to the browser (via [renderImage()](renderImage())), should the brush be reset? The default, FALSE, is useful if you want to update the plot while keeping the brush. Using TRUE is useful if you want to clear the brush whenever the plot is updated. |

---

| builder | *HTML Builder Functions* |
|---|---|

---

### Description

Simple functions for constructing HTML documents.

### Usage

```
tags

p(..., .noWS = NULL)

h1(..., .noWS = NULL)

h2(..., .noWS = NULL)

h3(..., .noWS = NULL)

h4(..., .noWS = NULL)

h5(..., .noWS = NULL)

h6(..., .noWS = NULL)

a(..., .noWS = NULL)
```

```
br(..., .noWS = NULL)

div(..., .noWS = NULL)

span(..., .noWS = NULL)

pre(..., .noWS = NULL)

code(..., .noWS = NULL)

img(..., .noWS = NULL)

strong(..., .noWS = NULL)

em(..., .noWS = NULL)

hr(..., .noWS = NULL)
```

## 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), raw HTML (see [HTML](#)), and `html_dependency` objects. You can also pass lists that contain tags, text nodes, or HTML. To use boolean attributes, use a named argument with a NA value. (see example) |
| `.noWS` | A character vector used to omit some of the whitespace that would normally be written around this tag. Valid options include `before`, `after`, `outside`, `after-begin`, and `before-end`. Any number of these options can be specified. |

## 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](#)().

## References

- W3C html specification about boolean attributes [https://www.w3.org/TR/html5/infrastructure.html#sec-boolean-attributes](https://www.w3.org/TR/html5/infrastructure.html#sec-boolean-attributes)

## 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))

# create an html5 audio tag with controls.
# controls is a boolean attributes
audio_tag <- tags$audio(
  controls = NA,
  tags$source(
    src = "myfile.wav",
    type = "audio/wav"
  )
)
cat(as.character(audio_tag))

# suppress the whitespace between tags
oneline <- tags$span(
  tags$strong("I'm strong", .noWS="outside")
)
cat(as.character(oneline))
```

---

callModule                     *Invoke a Shiny module*

---

## Description

Shiny's module feature lets you break complicated UI and server logic into smaller, self-contained pieces. Compared to large monolithic Shiny apps, modules are easier to reuse and easier to reason about. See the article at <http://shiny.rstudio.com/articles/modules.html> to learn more.

## Usage

```
callModule(module, id, ..., session = getDefaultReactiveDomain())
```

## Arguments

| | |
|---|---|
| module | A Shiny module server function |
| id | An ID string that corresponds with the ID used to call the module's UI function |
| ... | Additional parameters to pass to module server function |
| session | Session from which to make a child scope (the default should almost always be used) |

**Value**

The return value, if any, from executing the module server function

**See Also**

<http://shiny.rstudio.com/articles/modules.html>

---

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 = NULL, selected = NULL,
  inline = FALSE, width = NULL, choiceNames = NULL,
  choiceValues = NULL)
```

**Arguments**

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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. If this argument is provided, then choiceNames and choiceValues must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings. |
| selected | The values that should be initially selected, if any. |
| inline | If TRUE, render the choices inline (i.e. horizontally) |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| choiceNames, choiceValues | |
| | List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, choiceNames and choiceValues must have the same length). If either of these arguments is provided, then the other *must* be provided and choices *must not* be provided. The advantage of using both of these over a named list for choices is that choiceNames allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples. |

**Value**

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

**Server value**

Character vector of values corresponding to the boxes that are checked.

**See Also**

checkboxInput(), updateCheckboxGroupInput()

Other input elements: actionButton, checkboxInput, dateInput, dateRangeInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput, textInput, varSelectInput

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  checkboxGroupInput("variable", "Variables to show:",
                     c("Cylinders" = "cyl",
                       "Transmission" = "am",
                       "Gears" = "gear")),
  tableOutput("data")
)

server <- function(input, output, session) {
  output$data <- renderTable({
    mtcars[, c("mpg", input$variable), drop = FALSE]
  }, rownames = TRUE)
}

shinyApp(ui, server)

ui <- fluidPage(
  checkboxGroupInput("icons", "Choose icons:",
    choiceNames =
      list(icon("calendar"), icon("bed"),
           icon("cog"), icon("bug")),
    choiceValues =
      list("calendar", "bed", "cog", "bug")
  ),
  textOutput("txt")
)

server <- function(input, output, session) {
  output$txt <- renderText({
    icons <- paste(input$icons, collapse = ", ")
    paste("You chose", icons)
  })
}

shinyApp(ui, server)
}
```

---

checkboxInput                    *Checkbox Input Control*

---

### Description

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

### Usage

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

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| value | Initial value (TRUE or FALSE). |
| width | The width of the input, e.g. '400px', or '100%'; see validateCssUnit(). |

### Value

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

### Server value

TRUE if checked, FALSE otherwise.

### See Also

checkboxGroupInput(), updateCheckboxInput()

Other input elements: actionButton, checkboxGroupInput, dateInput, dateRangeInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput, textInput, varSelectInput

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  checkboxInput("somevalue", "Some value", FALSE),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({ input$somevalue })
}
shinyApp(ui, server)
}
```

---

clickOpts                    *Create an object representing click options*

---

### Description

This generates an object representing click options, to be passed as the click argument of [imageOutput()](#) or [plotOutput()](#).

### Usage

```
clickOpts(id = NULL, clip = TRUE)
```

### Arguments

| | |
|---|---|
| id | Input value name. For example, if the value is "plot_click", then the click coordinates will be available as input$plot_click. |
| clip | Should the click area be clipped to the plotting area? If FALSE, then the server will receive click events even when the mouse is outside the plotting area, as long as it is still inside the image. |

---

column                       *Create a column within a UI definition*

---

### Description

Create a column for use within a [fluidRow()](#) or [fixedRow()](#)

### Usage

```
column(width, ..., offset = 0)
```

### Arguments

| | |
|---|---|
| width | The grid width of the column (must be between 1 and 12) |
| ... | Elements to include within the column |
| offset | The number of columns to offset this column from the end of the previous column. |

### Value

A column that can be included within a [fluidRow()](#) or [fixedRow()](#).

### See Also

[fluidRow()](#), [fixedRow()](#).

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  fluidRow(
    column(4,
      sliderInput("obs", "Number of observations:",
                  min = 1, max = 1000, value = 500)
    ),
    column(8,
      plotOutput("distPlot")
    )
  )
)

server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}

shinyApp(ui, server)


ui <- fluidPage(
  fluidRow(
    column(width = 4,
      "4"
    ),
    column(width = 3, offset = 2,
      "3 offset 2"
    )
  )
)
shinyApp(ui, server = function(input, output) { })
}
```

---

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, ..., ns = NS(NULL))
```

## Arguments

| | |
|---|---|
| condition | A JavaScript expression that will be evaluated repeatedly to determine whether the panel should be displayed. |
| ... | Elements to include in the panel. |
| ns | The [namespace()](namespace()) object of the current module, if any. |

## 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.)

## Note

You are not recommended to use special JavaScript characters such as a period . in the input id's, but if you do use them anyway, for example, inputId = "foo.bar", you will have to use input["foo.bar"] instead of input.foo.bar to read the input value.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    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 = 50, value = 10)
        )
      )
    ),
    mainPanel(
      plotOutput("plot")
    )
  )

  server <- function(input, output) {
    x <- rnorm(100)
    y <- rnorm(100)
```

```
    output$plot <- renderPlot({
      if (input$plotType == "scatter") {
        plot(x, y)
      } else {
        breaks <- input$breaks
        if (breaks == "custom") {
          breaks <- input$breakCount
        }

        hist(x, breaks = breaks)
      }
    })
  }

  shinyApp(ui, server)
}
```

createRenderFunction *Implement render functions*

---

## Description

Implement render functions

## Usage

```
createRenderFunction(func, transform = function(value, session, name,
  ...) value, outputFunc = NULL, outputArgs = NULL)
```

## Arguments

func
: A function without parameters, that returns user data. If the returned value is a promise, then the render function will proceed in async mode.

transform
: A function that takes four arguments: value, session, name, and ... (for future-proofing). This function will be invoked each time a value is returned from func, and is responsible for changing the value into a JSON-ready value to be JSON-encoded and sent to the browser.

outputFunc
: The UI function that is used (or most commonly used) with this render function. This can be used in R Markdown documents to create complete output widgets out of just the render function.

outputArgs
: A list of arguments to pass to the outputFunc. Render functions should include outputArgs = list() in their own parameter list, and pass through the value as this argument, to allow app authors to customize outputs. (Currently, this is only supported for dynamically generated UIs, such as those created by Shiny code snippets embedded in R Markdown documents).

**Value**

An annotated render function, ready to be assigned to an `output` slot.

---

createWebDependency     *Create a web dependency*

---

**Description**

Ensure that a file-based HTML dependency (from the htmltools package) can be served over Shiny's HTTP server. This function works by using [addResourcePath()](addResourcePath()) to map the HTML dependency's directory to a URL.

**Usage**

```
createWebDependency(dependency, scrubFile = TRUE)
```

**Arguments**

| | |
|---|---|
| dependency | A single HTML dependency object, created using [htmltools::htmlDependency()](htmltools::htmlDependency()). If the `src` value is named, then `href` and/or `file` names must be present. |
| scrubFile | If TRUE (the default), remove `src$file` for the dependency. This prevents the local file path from being sent to the client when dynamic web dependencies are used. If FALSE, don't remove `src$file`. Setting it to FALSE should be needed only in very unusual cases. |

**Value**

A single HTML dependency object that has an `href`-named element in its `src`.

---

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", width = NULL, autoclose = TRUE,
  datesdisabled = NULL, daysofweekdisabled = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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 "ar", "az", "bg", "bs", "ca", "cs", "cy", "da", "de", "el", "en-AU", "en-GB", "eo", "es", "et", "eu", "fa", "fi", "fo", "fr-CH", "fr", "gl", "he", "hr", "hu", "hy", "id", "is", "it-CH", "it", "ja", "ka", "kh", "kk", "ko", "kr", "lt", "lv", "me", "mk", "mn", "ms", "nb", "nl-BE", "nl", "no", "pl", "pt-BR", "pt", "ro", "rs-latin", "rs", "ru", "sk", "sl", "sq", "sr-latin", "sr", "sv", "sw", "th", "tr", "uk", "vi", "zh-CN", and "zh-TW". |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| autoclose | Whether or not to close the datepicker immediately when a date is selected. |
| datesdisabled | Which dates should be disabled. Either a Date object, or a string in yyyy-mm-dd format. |
| daysofweekdisabled | |
| | Days of the week that should be disabled. Should be a integer vector with values from 0 (Sunday) to 6 (Saturday). |

## 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 (1-12)
- 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

**Server value**

A Date vector of length 1.

**See Also**

dateRangeInput(), updateDateInput()

Other input elements: actionButton, checkboxGroupInput, checkboxInput, dateRangeInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput, textInput, varSelectInput

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  dateInput("date1", "Date:", value = "2012-02-29"),

  # Default value is the date in client's time zone
  dateInput("date2", "Date:"),

  # value is always yyyy-mm-dd, even if the display format is different
  dateInput("date3", "Date:", value = "2012-02-29", format = "mm/dd/yy"),

  # Pass in a Date object
  dateInput("date4", "Date:", value = Sys.Date()-10),

  # Use different language and different first day of week
  dateInput("date5", "Date:",
          language = "ru",
          weekstart = 1),

  # Start with decade view instead of default month view
  dateInput("date6", "Date:",
            startview = "decade"),

  # Disable Mondays and Tuesdays.
  dateInput("date7", "Date:", daysofweekdisabled = c(1,2)),

  # Disable specific dates.
  dateInput("date8", "Date:", value = "2012-02-29",
            datesdisabled = c("2012-03-01", "2012-03-02"))
)

shinyApp(ui, server = function(input, output) { })
}
```

---

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 ", width = NULL,
  autoclose = TRUE)
```

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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 "ar", "az", "bg", "bs", "ca", "cs", "cy", "da", "de", "el", "en-AU", "en-GB", "eo", "es", "et", "eu", "fa", "fi", "fo", "fr-CH", "fr", "gl", "he", "hr", "hu", "hy", "id", "is", "it-CH", "it", "ja", "ka", "kh", "kk", "ko", "kr", "lt", "lv", "me", "mk", "mn", "ms", "nb", "nl-BE", "nl", "no", "pl", "pt-BR", "pt", "ro", "rs-latin", "rs", "ru", "sk", "sl", "sq", "sr-latin", "sr", "sv", "sw", "th", "tr", "uk", "vi", "zh-CN", and "zh-TW". |
| separator | String to display between the start and end input boxes. |
| width | The width of the input, e.g. '400px', or '100%'; see validateCssUnit(). |
| autoclose | Whether or not to close the datepicker immediately when a date is selected. |

**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 (1-12)
- 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

**Server value**

A Date vector of length 2.

**See Also**

dateInput(), updateDateRangeInput()

Other input elements: actionButton, checkboxGroupInput, checkboxInput, dateInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput, textInput, varSelectInput

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  dateRangeInput("daterange1", "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("daterange2", "Date range:"),

  # start and end are always specified in yyyy-mm-dd, even if the display
  # format is different
  dateRangeInput("daterange3", "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("daterange4", "Date range:",
                start = Sys.Date()-10,
                end = Sys.Date()+10),

    # Use different language and different first day of week
    dateRangeInput("daterange5", "Date range:",
                language = "de",
                weekstart = 1),

    # Start with decade view instead of default month view
    dateRangeInput("daterange6", "Date range:",
                startview = "decade")
)

shinyApp(ui, server = function(input, output) { })
}
```

dblclickOpts                *Create an object representing double-click options*

### Description

This generates an object representing dobule-click options, to be passed as the dblclick argument
of [imageOutput()](#) or [plotOutput()](#).

### Usage

```
dblclickOpts(id = NULL, clip = TRUE, delay = 400)
```

### Arguments

| | |
|---|---|
| id | Input value name. For example, if the value is "plot_dblclick", then the click coordinates will be available as input$plot_dblclick. |
| clip | Should the click area be clipped to the plotting area? If FALSE, then the server will receive double-click events even when the mouse is outside the plotting area, as long as it is still inside the image. |
| delay | Maximum delay (in ms) between a pair clicks for them to be counted as a double-click. |

---

debounce                     *Slow down a reactive expression with debounce/throttle*

---

### Description

Transforms a reactive expression by preventing its invalidation signals from being sent unnecessarily often. This lets you ignore a very "chatty" reactive expression until it becomes idle, which is useful when the intermediate values don't matter as much as the final value, and the downstream calculations that depend on the reactive expression take a long time. debounce and throttle use different algorithms for slowing down invalidation signals; see Details.

### Usage

```
debounce(r, millis, priority = 100,
  domain = getDefaultReactiveDomain())

throttle(r, millis, priority = 100,
  domain = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| r | A reactive expression (that invalidates too often). |
| millis | The debounce/throttle time window. You may optionally pass a no-arg function or reactive expression instead, e.g. to let the end-user control the time window. |
| priority | Debounce/throttle is implemented under the hood using observers. Use this parameter to set the priority of these observers. Generally, this should be higher than the priorities of downstream observers and outputs (which default to zero). |
| domain | See domains. |

### Details

This is not a true debounce/throttle in that it will not prevent r from being called many times (in fact it may be called more times than usual), but rather, the reactive invalidation signal that is produced by r is debounced/throttled instead. Therefore, these functions should be used when r is cheap but the things it will trigger (downstream outputs and reactives) are expensive.

Debouncing means that every invalidation from r will be held for the specified time window. If r invalidates again within that time window, then the timer starts over again. This means that as long as invalidations continually arrive from r within the time window, the debounced reactive will not invalidate at all. Only after the invalidations stop (or slow down sufficiently) will the downstream invalidation be sent.

ooo-oo-oo----=> -----------o-

(In this graphical depiction, each character represents a unit of time, and the time window is 3 characters.)

Throttling, on the other hand, delays invalidation if the *throttled* reactive recently (within the time window) invalidated. New `r` invalidations do not reset the time window. This means that if invalidations continually come from `r` within the time window, the throttled reactive will invalidate regularly, at a rate equal to or slower than than the time window.

```
ooo-oo-oo----=> o--o--o--o---
```

### Limitations

Because R is single threaded, we can't come close to guaranteeing that the timing of debounce/throttle (or any other timing-related functions in Shiny) will be consistent or accurate; at the time we want to emit an invalidation signal, R may be performing a different task and we have no way to interrupt it (nor would we necessarily want to if we could). Therefore, it's best to think of the time windows you pass to these functions as minimums.

You may also see undesirable behavior if the amount of time spent doing downstream processing for each change approaches or exceeds the time window: in this case, debounce/throttle may not have any effect, as the time each subsequent event is considered is already after the time window has expired.

### Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

library(shiny)
library(magrittr)

ui <- fluidPage(
  plotOutput("plot", click = clickOpts("hover")),
  helpText("Quickly click on the plot above, while watching the result table below:"),
  tableOutput("result")
)

server <- function(input, output, session) {
  hover <- reactive({
    if (is.null(input$hover))
      list(x = NA, y = NA)
    else
      input$hover
  })
  hover_d <- hover %>% debounce(1000)
  hover_t <- hover %>% throttle(1000)

  output$plot <- renderPlot({
    plot(cars)
  })

  output$result <- renderTable({
    data.frame(
      mode = c("raw", "throttle", "debounce"),
      x = c(hover()$x, hover_t()$x, hover_d()$x),
```

```
        y = c(hover()$y, hover_t()$y, hover_d()$y)
      )
    })
  }

  shinyApp(ui, server)
}
```

---

diskCache                   *Create a disk cache object*

---

### Description

A disk cache object is a key-value store that saves the values as files in a directory on disk. Objects can be stored and retrieved using the get() and set() methods. Objects are automatically pruned from the cache according to the parameters max_size, max_age, max_n, and evict.

### Usage

```
diskCache(dir = NULL, max_size = 10 * 1024^2, max_age = Inf,
  max_n = Inf, evict = c("lru", "fifo"), destroy_on_finalize = FALSE,
  missing = key_missing(), exec_missing = FALSE, logfile = NULL)
```

### Arguments

| | |
|---|---|
| dir | Directory to store files for the cache. If NULL (the default) it will create and use a temporary directory. |
| max_size | Maximum size of the cache, in bytes. If the cache exceeds this size, cached objects will be removed according to the value of the evict. Use Inf for no size limit. |
| max_age | Maximum age of files in cache before they are evicted, in seconds. Use Inf for no age limit. |
| max_n | Maximum number of objects in the cache. If the number of objects exceeds this value, then cached objects will be removed according to the value of evict. Use Inf for no limit of number of items. |
| evict | The eviction policy to use to decide which objects are removed when a cache pruning occurs. Currently, "lru" and "fifo" are supported. |
| destroy_on_finalize | |
| | If TRUE, then when the DiskCache object is garbage collected, the cache directory and all objects inside of it will be deleted from disk. If FALSE (the default), it will do nothing when finalized. |
| missing | A value to return or a function to execute when get(key) is called but the key is not present in the cache. The default is a [key_missing()](#) object. If it is a function to execute, the function must take one argument (the key), and you must also use exec_missing = TRUE. If it is a function, it is useful in most cases |

for it to throw an error, although another option is to return a value. If a value is returned, that value will in turn be returned by `get()`. See section Missing keys for more information.

exec_missing    If `FALSE` (the default), then treat `missing` as a value to return when `get()` results in a cache miss. If `TRUE`, treat `missing` as a function to execute when `get()` results in a cache miss.

logfile         An optional filename or connection object to where logging information will be written. To log to the console, use `stdout()`.

### Missing Keys

The `missing` and `exec_missing` parameters controls what happens when `get()` is called with a key that is not in the cache (a cache miss). The default behavior is to return a [key_missing()](#) object. This is a *sentinel value* that indicates that the key was not present in the cache. You can test if the returned value represents a missing key by using the [is.key_missing()](#) function. You can also have `get()` return a different sentinel value, like `NULL`. If you want to throw an error on a cache miss, you can do so by providing a function for `missing` that takes one argument, the key, and also use `exec_missing=TRUE`.

When the cache is created, you can supply a value for `missing`, which sets the default value to be returned for missing values. It can also be overridden when `get()` is called, by supplying a `missing` argument. For example, if you use `cache$get("mykey",missing = NULL)`, it will return `NULL` if the key is not in the cache.

If your cache is configured so that `get()` returns a sentinel value to represent a cache miss, then `set` will also not allow you to store the sentinel value in the cache. It will throw an error if you attempt to do so.

Instead of returning the same sentinel value each time there is cache miss, the cache can execute a function each time `get()` encounters missing key. If the function returns a value, then `get()` will in turn return that value. However, a more common use is for the function to throw an error. If an error is thrown, then `get()` will not return a value.

To do this, pass a one-argument function to `missing`, and use `exec_missing=TRUE`. For example, if you want to throw an error that prints the missing key, you could do this:

```
diskCache(
  missing = function(key) {
    stop("Attempted to get missing key: ", key)
  },
  exec_missing = TRUE
)
```

If you use this, the code that calls `get()` should be wrapped with [tryCatch()](#) to gracefully handle missing keys.

### Cache pruning

Cache pruning occurs when `set()` is called, or it can be invoked manually by calling `prune()`.

The disk cache will throttle the pruning so that it does not happen on every call to set(), because the filesystem operations for checking the status of files can be slow. Instead, it will prune once in every 20 calls to set(), or if at least 5 seconds have elapsed since the last prune occurred, whichever is first. These parameters are currently not customizable, but may be in the future.

When a pruning occurs, if there are any objects that are older than max_age, they will be removed.

The max_size and max_n parameters are applied to the cache as a whole, in contrast to max_age, which is applied to each object individually.

If the number of objects in the cache exceeds max_n, then objects will be removed from the cache according to the eviction policy, which is set with the evict parameter. Objects will be removed so that the number of items is max_n.

If the size of the objects in the cache exceeds max_size, then objects will be removed from the cache. Objects will be removed from the cache so that the total size remains under max_size. Note that the size is calculated using the size of the files, not the size of disk space used by the files — these two values can differ because of files are stored in blocks on disk. For example, if the block size is 4096 bytes, then a file that is one byte in size will take 4096 bytes on disk.

Another time that objects can be removed from the cache is when get() is called. If the target object is older than max_age, it will be removed and the cache will report it as a missing value.

**Eviction policies**

If max_n or max_size are used, then objects will be removed from the cache according to an eviction policy. The available eviction policies are:

"lru" Least Recently Used. The least recently used objects will be removed. This uses the filesystem's mtime property. When "lru" is used, each get() is called, it will update the file's mtime.

"fifo" First-in-first-out. The oldest objects will be removed.

Both of these policies use files' mtime. Note that some filesystems (notably FAT) have poor mtime resolution. (atime is not used because support for atime is worse than mtime.)

**Sharing among multiple processes**

The directory for a DiskCache can be shared among multiple R processes. To do this, each R process should have a DiskCache object that uses the same directory. Each DiskCache will do pruning independently of the others, so if they have different pruning parameters, then one DiskCache may remove cached objects before another DiskCache would do so.

Even though it is possible for multiple processes to share a DiskCache directory, this should not be done on networked file systems, because of slow performance of networked file systems can cause problems. If you need a high-performance shared cache, you can use one built on a database like Redis, SQLite, mySQL, or similar.

When multiple processes share a cache directory, there are some potential race conditions. For example, if your code calls exists(key) to check if an object is in the cache, and then call get(key), the object may be removed from the cache in between those two calls, and get(key) will throw an error. Instead of calling the two functions, it is better to simply call get(key), and use tryCatch() to handle the error that is thrown if the object is not in the cache. This effectively tests for existence and gets the object in one operation.

It is also possible for one processes to prune objects at the same time that another processes is trying to prune objects. If this happens, you may see a warning from `file.remove()` failing to remove a file that has already been deleted.

### Methods

A disk cache object has the following methods:

`get(key, missing, exec_missing)` Returns the value associated with key. If the key is not in the cache, then it returns the value specified by `missing` or, `missing` is a function and `exec_missing=TRUE`, then executes `missing`. The function can throw an error or return the value. If either of these parameters are specified here, then they will override the defaults that were set when the DiskCache object was created. See section Missing Keys for more information.

`set(key, value)` Stores the key-value pair in the cache.

`exists(key)` Returns `TRUE` if the cache contains the key, otherwise `FALSE`.

`size()` Returns the number of items currently in the cache.

`keys()` Returns a character vector of all keys currently in the cache.

`reset()` Clears all objects from the cache.

`destroy()` Clears all objects in the cache, and removes the cache directory from disk.

`prune()` Prunes the cache, using the parameters specified by `max_size`, `max_age`, `max_n`, and `evict`.

---

| domains | *Reactive domains* |
|---------|--------------------|

---

### Description

Reactive domains are a mechanism for establishing ownership over reactive primitives (like reactive expressions and observers), even if the set of reactive primitives is dynamically created. This is useful for lifetime management (i.e. destroying observers when the Shiny session that created them ends) and error handling.

### Usage

```
getDefaultReactiveDomain()

withReactiveDomain(domain, expr)

onReactiveDomainEnded(domain, callback, failIfNull = FALSE)
```

### Arguments

| | |
|---|---|
| domain | A valid domain object (for example, a Shiny session), or `NULL` |
| expr | An expression to evaluate under `domain` |
| callback | A callback function to be invoked |
| failIfNull | If `TRUE` then an error is given if the `domain` is `NULL` |

## Details

At any given time, there can be either a single "default" reactive domain object, or none (i.e. the reactive domain object is NULL). You can access the current default reactive domain by calling `getDefaultReactiveDomain`.

Unless you specify otherwise, newly created observers and reactive expressions will be assigned to the current default domain (if any). You can override this assignment by providing an explicit `domain` argument to `reactive()` or `observe()`.

For advanced usage, it's possible to override the default domain using `withReactiveDomain`. The `domain` argument will be made the default domain while `expr` is evaluated.

Implementers of new reactive primitives can use `onReactiveDomainEnded` as a convenience function for registering callbacks. If the reactive domain is NULL and `failIfNull` is FALSE, then the callback will never be invoked.

---

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. |
| ... | Other arguments to pass to the container tag function. |

## 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,
  outputArgs = list())
```

## 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. |
| outputArgs | A list of arguments to be passed through to the implicit call to downloadButton() when downloadHandler is used in an interactive R Markdown document. |

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  downloadLink("downloadData", "Download")
)

server <- function(input, output) {
  # Our dataset
  data <- mtcars

  output$downloadData <- downloadHandler(
    filename = function() {
      paste("data-", Sys.Date(), ".csv", sep="")
    },
    content = function(file) {
      write.csv(data, file)
    }
  )
}

shinyApp(ui, server)
}
```

---

| enableBookmarking | *Enable bookmarking for a Shiny application* |
|---|---|

---

## Description

There are two types of bookmarking: saving an application's state to disk on the server, and encoding the application's state in a URL. For state that has been saved to disk, the state can be restored with the corresponding state ID. For URL-encoded state, the state of the application is encoded in the URL, and no server-side storage is needed.

URL-encoded bookmarking is appropriate for applications where there not many input values that need to be recorded. Some browsers have a length limit for URLs of about 2000 characters, and if there are many inputs, the length of the URL can exceed that limit.

Saved-on-server bookmarking is appropriate when there are many inputs, or when the bookmarked state requires storing files.

## Usage

```
enableBookmarking(store = c("url", "server", "disable"))
```

## Arguments

store        Either "url", which encodes all of the relevant values in a URL, "server", which saves to disk on the server, or "disable", which disables any previously-enabled bookmarking.

**Details**

For restoring state to work properly, the UI must be a function that takes one argument, `request`. In most Shiny applications, the UI is not a function; it might have the form `fluidPage(....)`. Converting it to a function is as simple as wrapping it in a function, as in `function(request) { fluidPage(....) }`.

By default, all input values will be bookmarked, except for the values of passwordInputs. fileInputs will be saved if the state is saved on a server, but not if the state is encoded in a URL.

When bookmarking state, arbitrary values can be stored, by passing a function as the `onBookmark` argument. That function will be passed a `ShinySaveState` object. The `values` field of the object is a list which can be manipulated to save extra information. Additionally, if the state is being saved on the server, and the `dir` field of that object can be used to save extra information to files in that directory.

For saved-to-server state, this is how the state directory is chosen:

- If running in a hosting environment such as Shiny Server or Connect, the hosting environment will choose the directory.
- If running an app in a directory with [runApp()](), the saved states will be saved in a subdirectory of the app called shiny_bookmarks.
- If running a Shiny app object that is generated from code (not run from a directory), the saved states will be saved in a subdirectory of the current working directory called shiny_bookmarks.

When used with [shinyApp()](), this function must be called before `shinyApp()`, or in the `shinyApp()`'s `onStart` function. An alternative to calling the `enableBookmarking()` function is to use the `enableBookmarking` *argument* for `shinyApp()`. See examples below.

**See Also**

[onBookmark()](), [onBookmarked()](), [onRestore()](), and [onRestored()]() for registering callback functions that are invoked when the state is bookmarked or restored.

Also see [updateQueryString()]().

**Examples**

```
## Only run these examples in interactive R sessions
if (interactive()) {

# Basic example with state encoded in URL
ui <- function(request) {
  fluidPage(
    textInput("txt", "Text"),
    checkboxInput("chk", "Checkbox"),
    bookmarkButton()
  )
}
server <- function(input, output, session) { }
enableBookmarking("url")
shinyApp(ui, server)
```

```
# An alternative to calling enableBookmarking(): use shinyApp's
# enableBookmarking argument
shinyApp(ui, server, enableBookmarking = "url")


# Same basic example with state saved to disk
enableBookmarking("server")
shinyApp(ui, server)


# Save/restore arbitrary values
ui <- function(req) {
  fluidPage(
    textInput("txt", "Text"),
    checkboxInput("chk", "Checkbox"),
    bookmarkButton(),
    br(),
    textOutput("lastSaved")
  )
}
server <- function(input, output, session) {
  vals <- reactiveValues(savedTime = NULL)
  output$lastSaved <- renderText({
    if (!is.null(vals$savedTime))
      paste("Last saved at", vals$savedTime)
    else
      ""
  })

  onBookmark(function(state) {
    vals$savedTime <- Sys.time()
    # state is a mutable reference object, and we can add arbitrary values
    # to it.
    state$values$time <- vals$savedTime
  })
  onRestore(function(state) {
    vals$savedTime <- state$values$time
  })
}
enableBookmarking(store = "url")
shinyApp(ui, server)


# Usable with dynamic UI (set the slider, then change the text input,
# click the bookmark button)
ui <- function(request) {
  fluidPage(
    sliderInput("slider", "Slider", 1, 100, 50),
    uiOutput("ui"),
    bookmarkButton()
  )
}
server <- function(input, output, session) {
```

```
  output$ui <- renderUI({
    textInput("txt", "Text", input$slider)
  })
}
enableBookmarking("url")
shinyApp(ui, server)


# Exclude specific inputs (The only input that will be saved in this
# example is chk)
ui <- function(request) {
  fluidPage(
    passwordInput("pw", "Password"), # Passwords are never saved
    sliderInput("slider", "Slider", 1, 100, 50), # Manually excluded below
    checkboxInput("chk", "Checkbox"),
    bookmarkButton()
  )
}
server <- function(input, output, session) {
  setBookmarkExclude("slider")
}
enableBookmarking("url")
shinyApp(ui, server)


# Update the browser's location bar every time an input changes. This should
# not be used with enableBookmarking("server"), because that would create a
# new saved state on disk every time the user changes an input.
ui <- function(req) {
  fluidPage(
    textInput("txt", "Text"),
    checkboxInput("chk", "Checkbox")
  )
}
server <- function(input, output, session) {
  observe({
    # Trigger this observer every time an input changes
    reactiveValuesToList(input)
    session$doBookmark()
  })
  onBookmarked(function(url) {
    updateQueryString(url)
  })
}
enableBookmarking("url")
shinyApp(ui, server)


# Save/restore uploaded files
ui <- function(request) {
  fluidPage(
    sidebarLayout(
      sidebarPanel(
```

```
        fileInput("file1", "Choose CSV File", multiple = TRUE,
          accept = c(
            "text/csv",
            "text/comma-separated-values,text/plain",
            ".csv"
          )
        ),
        tags$hr(),
        checkboxInput("header", "Header", TRUE),
        bookmarkButton()
      ),
      mainPanel(
        tableOutput("contents")
      )
    )
  )
}
server <- function(input, output) {
  output$contents <- renderTable({
    inFile <- input$file1
    if (is.null(inFile))
      return(NULL)

    if (nrow(inFile) == 1) {
      read.csv(inFile$datapath, header = input$header)
    } else {
      data.frame(x = "multiple files")
    }
  })
}
enableBookmarking("server")
shinyApp(ui, server)

}
```

---

exportTestValues          *Register expressions for export in test mode*

---

### Description

This function registers expressions that will be evaluated when a test export event occurs. These events are triggered by accessing a snapshot URL.

### Usage

```
exportTestValues(..., quoted_ = FALSE, env_ = parent.frame(),
  session_ = getDefaultReactiveDomain())
```

## Arguments

| | |
|---|---|
| `...` | Named arguments that are quoted or unquoted expressions that will be captured and evaluated when snapshot URL is visited. |
| `quoted_` | Are the expression quoted? Default is `FALSE`. |
| `env_` | The environment in which the expression should be evaluated. |
| `session_` | A Shiny session object. |

## Details

This function only has an effect if the app is launched in test mode. This is done by calling `runApp()` with `test.mode=TRUE`, or by setting the global option `shiny.testmode` to `TRUE`.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {

options(shiny.testmode = TRUE)

# This application shows the test snapshot URL; clicking on it will
# fetch the input, output, and exported values in JSON format.
shinyApp(
  ui = basicPage(
    h4("Snapshot URL: "),
    uiOutput("url"),
    h4("Current values:"),
    verbatimTextOutput("values"),
    actionButton("inc", "Increment x")
  ),

  server = function(input, output, session) {
    vals <- reactiveValues(x = 1)
    y <- reactive({ vals$x + 1 })

    observeEvent(input$inc, {
      vals$x <<- vals$x + 1
    })

    exportTestValues(
      x = vals$x,
      y = y()
    )

    output$url <- renderUI({
      url <- session$getTestSnapshotUrl(format="json")
      a(href = url, url)
    })

    output$values <- renderText({
      paste0("vals$x: ", vals$x, "\ny: ", y())
    })
```

```
    }
  )
}
```

---

exprToFunction                    *Convert an 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(), 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.

### Usage

```
fileInput(inputId, label, multiple = FALSE, accept = NULL,
  width = NULL, buttonLabel = "Browse...",
  placeholder = "No file selected")
```

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| multiple | Whether the user should be allowed to select and upload multiple files at once. **Does not work on older browsers, including Internet Explorer 9 and earlier.** |
| accept | A character vector of MIME types; gives the browser a hint of what kind of files the server is expecting. |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| buttonLabel | The label used on the button. Can be text or an HTML tag object. |
| placeholder | The text to show before a file has been uploaded. |

### Details

Whenever a file upload completes, the corresponding input variable is set to a dataframe. See the Server value section.

**Server value**

A data.frame that contains one row for each selected file, and 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.

## See Also

Other input elements: actionButton, checkboxGroupInput, checkboxInput, dateInput, dateRangeInput,
numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput,
textInput, varSelectInput

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      fileInput("file1", "Choose CSV File",
        accept = c(
          "text/csv",
          "text/comma-separated-values,text/plain",
          ".csv")
        ),
      tags$hr(),
      checkboxInput("header", "Header", TRUE)
    ),
    mainPanel(
      tableOutput("contents")
    )
  )
)

server <- function(input, output) {
  output$contents <- renderTable({
    # input$file1 will be NULL initially. After the user selects
    # and uploads a file, it will be a data frame with 'name',
    # 'size', 'type', and 'datapath' columns. The 'datapath'
    # column will contain the local filenames where the data can
    # be found.
    inFile <- input$file1

    if (is.null(inFile))
```

```
      return(NULL)

    read.csv(inFile$datapath, header = input$header)
  })
}

shinyApp(ui, server)
}
```

---

fillPage                        *Create a page that fills the window*

---

### Description

`fillPage` creates a page whose height and width always fill the available area of the browser window.

### Usage

```
fillPage(..., padding = 0, title = NULL, bootstrap = TRUE,
  theme = NULL)
```

### Arguments

| | |
|---|---|
| `...` | Elements to include within the page. |
| `padding` | Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively. |
| `title` | The title to use for the browser window/tab (it will not be shown in the document). |
| `bootstrap` | If `TRUE`, load the Bootstrap CSS library. |
| `theme` | URL to alternative Bootstrap stylesheet. |

### Details

The [`fluidPage()`](fluidPage) and [`fixedPage()`](fixedPage) functions are used for creating web pages that are laid out from the top down, leaving whitespace at the bottom if the page content's height is smaller than the browser window, and scrolling if the content is larger than the window.

`fillPage` is designed to latch the document body's size to the size of the window. This makes it possible to fill it with content that also scales to the size of the window.

For example, `fluidPage(plotOutput("plot",height = "100%"))` will not work as expected; the plot element's effective height will be 0, because the plot's containing elements (<div> and

<body>) have *automatic* height; that is, they determine their own height based on the height of their contained elements. However, fillPage(plotOutput("plot",height = "100%")) will work because fillPage fixes the <body> height at 100% of the window height.

Note that fillPage(plotOutput("plot")) will not cause the plot to fill the page. Like most Shiny output widgets, plotOutput's default height is a fixed number of pixels. You must explicitly set height = "100%" if you want a plot (or htmlwidget, say) to fill its container.

One must be careful what layouts/panels/elements come between the fillPage and the plots/widgets. Any container that has an automatic height will cause children with height = "100%" to misbehave. Stick to functions that are designed for fill layouts, such as the ones in this package.

### See Also

Other layout functions: fixedPage, flowLayout, fluidPage, navbarPage, sidebarLayout, splitLayout, verticalLayout

### Examples

```
fillPage(
  tags$style(type = "text/css",
    ".half-fill { width: 50%; height: 100%; }",
    "#one { float: left; background-color: #ddddff; }",
    "#two { float: right; background-color: #ccffcc; }"
  ),
  div(id = "one", class = "half-fill",
    "Left half"
  ),
  div(id = "two", class = "half-fill",
    "Right half"
  ),
  padding = 10
)

fillPage(
  fillRow(
    div(style = "background-color: red; width: 100%; height: 100%;"),
    div(style = "background-color: blue; width: 100%; height: 100%;")
  )
)
```

---

fillRow                         *Flex Box-based row/column layouts*

---

### Description

Creates row and column layouts with proportionally-sized cells, using the Flex Box layout model of CSS3. These can be nested to create arbitrary proportional-grid layouts. **Warning:** Flex Box is not well supported by Internet Explorer, so these functions should only be used where modern browsers can be assumed.

## Usage

```
fillRow(..., flex = 1, width = "100%", height = "100%")

fillCol(..., flex = 1, width = "100%", height = "100%")
```

## Arguments

| | |
|---|---|
| `...` | UI objects to put in each row/column cell; each argument will occupy a single cell. (To put multiple items in a single cell, you can use [tagList()](#) or [div()](#) to combine them.) Named arguments will be used as attributes on the div element that encapsulates the row/column. |
| `flex` | Determines how space should be distributed to the cells. Can be a single value like 1 or 2 to evenly distribute the available space; or use a vector of numbers to specify the proportions. For example, `flex = c(2,3)` would cause the space to be split 40%/60% between two cells. NA values will cause the corresponding cell to be sized according to its contents (without growing or shrinking). |
| `width, height` | The total amount of width and height to use for the entire row/column. For the default height of `"100%"` to be effective, the parent must be `fillPage`, another `fillRow`/`fillCol`, or some other HTML element whose height is not determined by the height of its contents. |

## Details

If you try to use `fillRow` and `fillCol` inside of other Shiny containers, such as [sidebarLayout()](#), [navbarPage()](#), or even `tags$div`, you will probably find that they will not appear. This is due to `fillRow` and `fillCol` defaulting to `height="100%"`, which will only work inside of containers that have determined their own size (rather than shrinking to the size of their contents, as is usually the case in HTML).

To avoid this problem, you have two options:

- only use fillRow/fillCol inside of `fillPage`, `fillRow`, or `fillCol`
- provide an explicit `height` argument to `fillRow`/`fillCol`

## Examples

```
# Only run this example in interactive R sessions.
if (interactive()) {

ui <- fillPage(fillRow(
  plotOutput("plotLeft", height = "100%"),
  fillCol(
    plotOutput("plotTopRight", height = "100%"),
    plotOutput("plotBottomRight", height = "100%")
  )
))

server <- function(input, output, session) {
  output$plotLeft <- renderPlot(plot(cars))
  output$plotTopRight <- renderPlot(plot(pressure))
```

```
    output$plotBottomRight <- renderPlot(plot(AirPassengers))
}

shinyApp(ui, server)

}
```

---

fixedPage                    *Create a page with a fixed layout*

---

### Description

Functions for creating fixed page layouts. A fixed page layout consists of rows which in turn include columns. Rows exist for the purpose of making sure their elements appear on the same line (if the browser has adequate width). Columns exist for the purpose of defining how much horizontal space within a 12-unit wide grid it's elements should occupy. Fixed pages limit their width to 940 pixels on a typical display, and 724px or 1170px on smaller and larger displays respectively.

### Usage

```
fixedPage(..., title = NULL, responsive = NULL, theme = NULL)

fixedRow(...)
```

### Arguments

| | |
|---|---|
| ... | Elements to include within the container |
| title | The browser window title (defaults to the host URL of the page) |
| responsive | This option is deprecated; it is no longer optional with Bootstrap 3. |
| theme | Alternative Bootstrap stylesheet (normally a css file within the www directory). For example, to use the theme located at www/bootstrap.css you would use theme = "bootstrap.css". |

### Details

To create a fixed page use the fixedPage function and include instances of fixedRow and [column()](column()) within it. Note that unlike [fluidPage()](fluidPage()), fixed pages cannot make use of higher-level layout functions like sidebarLayout, rather, all layout must be done with fixedRow and column.

### Value

A UI defintion that can be passed to the [shinyUI](shinyUI) function.

### Note

See the [Shiny Application Layout Guide](Shiny Application Layout Guide) for additional details on laying out fixed pages.

## See Also

[column()](column())

Other layout functions: [fillPage](fillPage), [flowLayout](flowLayout), [fluidPage](fluidPage), [navbarPage](navbarPage), [sidebarLayout](sidebarLayout), [splitLayout](splitLayout),
[verticalLayout](verticalLayout)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fixedPage(
  title = "Hello, Shiny!",
  fixedRow(
    column(width = 4,
      "4"
    ),
    column(width = 3, offset = 2,
      "3 offset 2"
    )
  )
)

shinyApp(ui, server = function(input, output) { })
}
```

---

flowLayout                          *Flow layout*

---

## Description

Lays out elements in a left-to-right, top-to-bottom arrangement. The elements on a given row will
be top-aligned with each other. This layout will not work well with elements that have a percentage-
based width (e.g. [plotOutput()](plotOutput()) at its default setting of width = "100%").

## Usage

```
flowLayout(..., cellArgs = list())
```

## Arguments

| | |
|---|---|
| ... | Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag. |
| cellArgs | Any additional attributes that should be used for each cell of the layout. |

## See Also

Other layout functions: [fillPage](fillPage), [fixedPage](fixedPage), [fluidPage](fluidPage), [navbarPage](navbarPage), [sidebarLayout](sidebarLayout), [splitLayout](splitLayout),
[verticalLayout](verticalLayout)

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- flowLayout(
  numericInput("rows", "How many rows?", 5),
  selectInput("letter", "Which letter?", LETTERS),
  sliderInput("value", "What value?", 0, 100, 50)
)
shinyApp(ui, server = function(input, output) { })
}
```

---

fluidPage                    *Create a page with fluid layout*

---

### Description

Functions for creating fluid page layouts. A fluid page layout consists of rows which in turn include columns. Rows exist for the purpose of making sure their elements appear on the same line (if the browser has adequate width). Columns exist for the purpose of defining how much horizontal space within a 12-unit wide grid it's elements should occupy. Fluid pages scale their components in realtime to fill all available browser width.

### Usage

```
fluidPage(..., title = NULL, responsive = NULL, theme = NULL)

fluidRow(...)
```

### Arguments

| | |
|---|---|
| ... | Elements to include within the page |
| title | The browser window title (defaults to the host URL of the page). Can also be set as a side effect of the [titlePanel()](#) function. |
| responsive | This option is deprecated; it is no longer optional with Bootstrap 3. |
| theme | Alternative Bootstrap stylesheet (normally a css file within the www directory). For example, to use the theme located at www/bootstrap.css you would use theme = "bootstrap.css". |

### Details

To create a fluid page use the fluidPage function and include instances of fluidRow and [column()](#) within it. As an alternative to low-level row and column functions you can also use higher-level layout functions like [sidebarLayout()](#).

### Value

A UI defintion that can be passed to the [shinyUI](#) function.

**Note**

See the Shiny-Application-Layout-Guide for additional details on laying out fluid pages.

**See Also**

column()

Other layout functions: fillPage, fixedPage, flowLayout, navbarPage, sidebarLayout, splitLayout, verticalLayout

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

# Example of UI with fluidPage
ui <- fluidPage(

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

  sidebarLayout(

    # 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")
    )
  )
)

# Server logic
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}

# Complete app with UI and server components
shinyApp(ui, server)


# UI demonstrating column layouts
ui <- fluidPage(
```

```
    title = "Hello Shiny!",
    fluidRow(
      column(width = 4,
        "4"
      ),
      column(width = 3, offset = 2,
        "3 offset 2"
      )
    )
  )

  shinyApp(ui, server = function(input, output) { })
}
```

---

freezeReactiveVal          *Freeze a reactive value*

---

### Description

These functions freeze a [reactiveVal()](#), or an element of a [reactiveValues()](#). If the value is
accessed while frozen, a "silent" exception is raised and the operation is stopped. This is the same
thing that happens if req(FALSE) is called. The value is thawed (un-frozen; accessing it will no
longer raise an exception) when the current reactive domain is flushed. In a Shiny application, this
occurs after all of the observers are executed.

### Usage

```
freezeReactiveVal(x)

freezeReactiveValue(x, name)
```

### Arguments

| | |
|---|---|
| x | For freezeReactiveValue, a [reactiveValues()](#) object (like input); for freezeReactiveVal, a [reactiveVal()](#) object. |
| name | The name of a value in the [reactiveValues()](#) object. |

### See Also

[req()](#)

### Examples

```
## Only run this examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  selectInput("data", "Data Set", c("mtcars", "pressure")),
  checkboxGroupInput("cols", "Columns (select 2)", character(0)),
```

```
    plotOutput("plot")
  )

  server <- function(input, output, session) {
    observe({
      data <- get(input$data)
      # Sets a flag on input$cols to essentially do req(FALSE) if input$cols
      # is accessed. Without this, an error will momentarily show whenever a
      # new data set is selected.
      freezeReactiveValue(input, "cols")
      updateCheckboxGroupInput(session, "cols", choices = names(data))
    })

    output$plot <- renderPlot({
      # When a new data set is selected, input$cols will have been invalidated
      # above, and this will essentially do the same as req(FALSE), causing
      # this observer to stop and raise a silent exception.
      cols <- input$cols
      data <- get(input$data)

      if (length(cols) == 2) {
        plot(data[[ cols[1] ]], data[[ cols[2] ]])
      }
    })
  }

  shinyApp(ui, server)
}
```

getCurrentOutputInfo    *Get information about the output that is currently being executed.*

### Description

Get information about the output that is currently being executed.

### Usage

```
getCurrentOutputInfo(session = getDefaultReactiveDomain())
```

### Arguments

session         The current Shiny session.

---

getQueryString                    *Get the query string / hash component from the URL*

---

### Description

Two user friendly wrappers for getting the query string and the hash component from the app's URL.

### Usage

```
getQueryString(session = getDefaultReactiveDomain())

getUrlHash(session = getDefaultReactiveDomain())
```

### Arguments

session          A Shiny session object.

### Details

These can be particularly useful if you want to display different content depending on the values in the query string / hash (e.g. instead of basing the conditional on an input or a calculated reactive, you can base it on the query string). However, note that, if you're changing the query string / hash programatically from within the server code, you must use updateQueryString(_yourNewQueryString_,mode = "push"). The default mode for updateQueryString is "replace", which doesn't raise any events, so any observers or reactives that depend on it will *not* get triggered. However, if you're changing the query string / hash directly by typing directly in the browser and hitting enter, you don't have to worry about this.

### Value

For getQueryString, a named list. For example, the query string ?param1=value1&param2=value2 becomes list(param1 = value1,param2 = value2). For getUrlHash, a character vector with the hash (including the leading # symbol).

### See Also

[updateQueryString()](updateQueryString())

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {

  ## App 1: getQueryString
  ## Printing the value of the query string
  ## (Use the back and forward buttons to see how the browser
  ## keeps a record of each state)
```

```
shinyApp(
  ui = fluidPage(
    textInput("txt", "Enter new query string"),
    helpText("Format: ?param1=val1&param2=val2"),
    actionButton("go", "Update"),
    hr(),
    verbatimTextOutput("query")
  ),
  server = function(input, output, session) {
    observeEvent(input$go, {
      updateQueryString(input$txt, mode = "push")
    })
    output$query <- renderText({
      query <- getQueryString()
      queryText <- paste(names(query), query,
                        sep = "=", collapse=", ")
      paste("Your query string is:\n", queryText)
    })
  }
)

## App 2: getUrlHash
## Printing the value of the URL hash
## (Use the back and forward buttons to see how the browser
## keeps a record of each state)
shinyApp(
  ui = fluidPage(
    textInput("txt", "Enter new hash"),
    helpText("Format: #hash"),
    actionButton("go", "Update"),
    hr(),
    verbatimTextOutput("hash")
  ),
  server = function(input, output, session) {
    observeEvent(input$go, {
      updateQueryString(input$txt, mode = "push")
    })
    output$hash <- renderText({
      hash <- getUrlHash()
      paste("Your hash is:\n", hash)
    })
  }
)
}
```

---

getShinyOption                    *Get or set Shiny options*

---

**Description**

getShinyOption() retrieves the value of a Shiny option. shinyOptions() sets the value of Shiny options; it can also be used to return a list of all currently-set Shiny options.

**Usage**

```
getShinyOption(name, default = NULL)

shinyOptions(...)
```

**Arguments**

| | |
|---|---|
| name | Name of an option to get. |
| default | Value to be returned if the option is not currently set. |
| ... | Options to set, with the form name = value. |

**Scope**

There is a global option set which is available by default. When a Shiny application is run with runApp(), that option set is duplicated and the new option set is available for getting or setting values. If options are set from global.R, app.R, ui.R, or server.R, or if they are set from inside the server function, then the options will be scoped to the application. When the application exits, the new option set is discarded and the global option set is restored.

**Options**

There are a number of global options that affect Shiny's behavior. These can be set globally with options() or locally (for a single app) with shinyOptions().

**shiny.autoreload (defaults to** FALSE**)** If TRUE when a Shiny app is launched, the app directory will be continually monitored for changes to files that have the extensions: r, htm, html, js, css, png, jpg, jpeg, gif. If any changes are detected, all connected Shiny sessions are reloaded. This allows for fast feedback loops when tweaking Shiny UI.

```
Since monitoring for changes is expensive (we simply poll for last
modified times), this feature is intended only for development.
```

```
You can customize the file patterns Shiny will monitor by setting the
shiny.autoreload.pattern option. For example, to monitor only ui.R:
`options(shiny.autoreload.pattern = glob2rx("ui.R"))`
```

```
The default polling interval is 500 milliseconds. You can change this
by setting e.g. `options(shiny.autoreload.interval = 2000)` (every
two seconds).
```

**shiny.deprecation.messages (defaults to** TRUE**)** This controls whether messages for deprecated functions in Shiny will be printed. See shinyDeprecated() for more information.

**shiny.error (defaults to** NULL**)** This can be a function which is called when an error occurs. For example, options(shiny.error=recover) will result a the debugger prompt when an error occurs.

**shiny.fullstacktrace (defaults to** FALSE**)** Controls whether "pretty" (FALSE) or full stack traces (TRUE) are dumped to the console when errors occur during Shiny app execution. Pretty stack traces attempt to only show user-supplied code, but this pruning can't always be done 100% correctly.

**shiny.host (defaults to** "127.0.0.1"**)** The IP address that Shiny should listen on. See runApp() for more information.

**shiny.jquery.version (defaults to** 3**)** The major version of jQuery to use. Currently only values of 3 or 1 are supported. If 1, then jQuery 1.12.4 is used. If 3, then jQuery 3.4.1 is used.

**shiny.json.digits (defaults to** 16**)** The number of digits to use when converting numbers to JSON format to send to the client web browser.

**shiny.launch.browser (defaults to** interactive()**)** A boolean which controls the default behavior when an app is run. See runApp() for more information.

**shiny.maxRequestSize (defaults to 5MB)** This is a number which specifies the maximum web request size, which serves as a size limit for file uploads.

**shiny.minified (defaults to** TRUE**)** By default Whether or not to include Shiny's JavaScript as a minified (shiny.min.js) or un-minified (shiny.js) file. The un-minified version is larger, but can be helpful for development and debugging.

**shiny.port (defaults to a random open port)** A port number that Shiny will listen on. See runApp() for more information.

**shiny.reactlog (defaults to** FALSE**)** If TRUE, enable logging of reactive events, which can be viewed later with the reactlogShow() function. This incurs a substantial performance penalty and should not be used in production.

**shiny.sanitize.errors (defaults to** FALSE**)** If TRUE, then normal errors (i.e. errors not wrapped in safeError) won't show up in the app; a simple generic error message is printed instead (the error and strack trace printed to the console remain unchanged). If you want to sanitize errors in general, but you DO want a particular error e to get displayed to the user, then set this option to TRUE and use stop(safeError(e)) for errors you want the user to see.

**shiny.stacktraceoffset (defaults to** TRUE**)** If TRUE, then Shiny's printed stack traces will display srcrefs one line above their usual location. This is an arguably more intuitive arrangement for casual R users, as the name of a function appears next to the srcref where it is defined, rather than where it is currently being called from.

**shiny.suppressMissingContextError (defaults to** FALSE**)** Normally, invoking a reactive outside of a reactive context (or isolate()) results in an error. If this is TRUE, don't error in these cases. This should only be used for debugging or demonstrations of reactivity at the console.

**shiny.testmode (defaults to** FALSE**)** If TRUE, then various features for testing Shiny applications are enabled.

**shiny.trace (defaults to** FALSE**)** Print messages sent between the R server and the web browser client to the R console. This is useful for debugging. Possible values are "send" (only print messages sent to the client), "recv" (only print messages received by the server), TRUE (print all messages), or FALSE (default; don't print any of these messages).

**shiny.usecairo (defaults to** TRUE**)** This is used to disable graphical rendering by the Cairo package, if it is installed. See plotPNG() for more information.

## Examples

```
## Not run:
shinyOptions(myOption = 10)
getShinyOption("myOption")

## End(Not run)
```

---

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.")
```

---

hoverOpts                   *Create an object representing hover options*

---

## Description

This generates an object representing hovering options, to be passed as the hover argument of [imageOutput()](imageOutput()) or [plotOutput()](plotOutput()).

## Usage

```
hoverOpts(id = NULL, delay = 300, delayType = c("debounce",
  "throttle"), clip = TRUE, nullOutside = TRUE)
```

## Arguments

| | |
|---|---|
| id | Input value name. For example, if the value is `"plot_hover"`, then the hover coordinates will be available as input$plot_hover. |
| delay | How long to delay (in milliseconds) when debouncing or throttling, before sending the mouse location to the server. |
| delayType | The type of algorithm for limiting the number of hover events. Use `"throttle"` to limit the number of hover events to one every `delay` milliseconds. Use `"debounce"` to suspend events while the cursor is moving, and wait until the cursor has been at rest for `delay` milliseconds before sending an event. |
| clip | Should the hover area be clipped to the plotting area? If FALSE, then the server will receive hover events even when the mouse is outside the plotting area, as long as it is still inside the image. |
| nullOutside | If `TRUE` (the default), the value will be set to `NULL` when the mouse exits the plotting area. If `FALSE`, the value will stop changing when the cursor exits the plotting area. |

---

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, inline = FALSE, container = if (inline) span else
  div, ...)

uiOutput(outputId, inline = FALSE, container = if (inline) span else
  div, ...)
```

### Arguments

| | |
|---|---|
| `outputId` | output variable to read the value from |
| `inline` | use an inline (`span()`) or block container (`div()`) for the output |
| `container` | a function to generate an HTML element to contain the text |
| `...` | Other arguments to pass to the container tag function. This is useful for providing additional classes for the tag. |

### 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")

# Using a custom container and class
tags$ul(
  htmlOutput("summary", container = tags$li, class = "custom-li-output")
)
```

---

htmlTemplate                    *Process an HTML template*

---

### Description

Process an HTML template and return a tagList object. If the template is a complete HTML document, then the returned object will also have class html_document, and can be passed to the function [renderDocument](#) to get the final HTML text.

### Usage

```
htmlTemplate(filename = NULL, ..., text_ = NULL, document_ = "auto")
```

### Arguments

| | |
|---|---|
| filename | Path to an HTML template file. Incompatible with text_. |
| ... | Variable values to use when processing the template. |
| text_ | A string to use as the template, instead of a file. Incompatible with filename. |
| document_ | Is this template a complete HTML document (TRUE), or a fragment of HTML that is to be inserted into an HTML document (FALSE)? With "auto" (the default), auto-detect by searching for the string "<HTML>" within the template. |

### See Also

[renderDocument](#)

---

icon                            *Create an icon*

---

### Description

Create an icon for use within a page. Icons can appear on their own, inside of a button, or as an icon for a [tabPanel()](#) within a [navbarPage()](#).

### Usage

```
icon(name, class = NULL, lib = "font-awesome")
```

### Arguments

| | |
|---|---|
| name | Name of icon. Icons are drawn from the [Font Awesome Free](#) (currently icons from the v5.3.1 set are supported with the v4 naming convention) and [Glyphicons](#) libraries. Note that the "fa-" and "glyphicon-" prefixes should not be used in icon names (i.e. the "fa-calendar" icon should be referred to as "calendar") |
| class | Additional classes to customize the style of the icon (see the [usage examples](#) for details on supported styles). |
| lib | Icon library to use ("font-awesome" or "glyphicon") |

## Value

An icon element

## See Also

For lists of available icons, see http://fontawesome.io/icons/ and http://getbootstrap.com/components/#glyphicons.

## Examples

```
# add an icon to a submit button
submitButton("Update View", icon = icon("refresh"))

navbarPage("App Title",
  tabPanel("Plot", icon = icon("bar-chart-o")),
  tabPanel("Summary", icon = icon("list-alt")),
  tabPanel("Table", icon = icon("table"))
)
```

---

include                         *Include Content From a File*

---

## Description

Load HTML, text, or rendered Markdown from a file and turn into HTML.

## 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.

---

inputPanel                          *Input panel*

---

**Description**

A [flowLayout()](#) with a grey border and light grey background, suitable for wrapping inputs.

**Usage**

```
inputPanel(...)
```

**Arguments**

...                  Input controls or other HTML elements.

---

insertTab                          *Dynamically insert/remove a tabPanel*

---

**Description**

Dynamically insert or remove a [tabPanel()](#) (or a [navbarMenu()](#)) from an existing [tabsetPanel()](#), [navlistPanel()](#) or [navbarPage()](#).

**Usage**

```
insertTab(inputId, tab, target, position = c("before", "after"),
  select = FALSE, session = getDefaultReactiveDomain())

prependTab(inputId, tab, select = FALSE, menuName = NULL,
  session = getDefaultReactiveDomain())

appendTab(inputId, tab, select = FALSE, menuName = NULL,
  session = getDefaultReactiveDomain())

removeTab(inputId, target, session = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| inputId | The id of the tabsetPanel (or navlistPanel or navbarPage) into which tab will be inserted/removed. |
| tab | The item to be added (must be created with tabPanel, or with navbarMenu). |
| target | If inserting: the value of an existing tabPanel, next to which tab will be added. If removing: the value of the tabPanel that you want to remove. See Details if you want to insert next to/remove an entire navbarMenu instead. |
| position | Should tab be added before or after the target tab? |
| select | Should tab be selected upon being inserted? |
| session | The shiny session within which to call this function. |
| menuName | This argument should only be used when you want to prepend (or append) tab to the beginning (or end) of an existing [navbarMenu()](#) (which must itself be part of an existing [navbarPage()](#)). In this case, this argument should be the menuName that you gave your navbarMenu when you first created it (by default, this is equal to the value of the title argument). Note that you still need to set the inputId argument to whatever the id of the parent navbarPage is. If menuName is left as NULL, tab will be prepended (or appended) to whatever inputId is. |

### Details

When you want to insert a new tab before or after an existing tab, you should use insertTab. When you want to prepend a tab (i.e. add a tab to the beginning of the tabsetPanel), use prependTab. When you want to append a tab (i.e. add a tab to the end of the tabsetPanel), use appendTab.

For navbarPage, you can insert/remove conventional tabPanels (whether at the top level or nested inside a navbarMenu), as well as an entire [navbarMenu()](#). For the latter case, target should be the menuName that you gave your navbarMenu when you first created it (by default, this is equal to the value of the title argument).

### See Also

[showTab()](#)

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {

# example app for inserting/removing a tab
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      actionButton("add", "Add 'Dynamic' tab"),
      actionButton("remove", "Remove 'Foo' tab")
    ),
    mainPanel(
      tabsetPanel(id = "tabs",
        tabPanel("Hello", "This is the hello tab"),
        tabPanel("Foo", "This is the foo tab"),
```

```r
        tabPanel("Bar", "This is the bar tab")
      )
    )
  )
)
server <- function(input, output, session) {
  observeEvent(input$add, {
    insertTab(inputId = "tabs",
      tabPanel("Dynamic", "This a dynamically-added tab"),
      target = "Bar"
    )
  })
  observeEvent(input$remove, {
    removeTab(inputId = "tabs", target = "Foo")
  })
}

shinyApp(ui, server)


# example app for prepending/appending a navbarMenu
ui <- navbarPage("Navbar page", id = "tabs",
  tabPanel("Home",
    actionButton("prepend", "Prepend a navbarMenu"),
    actionButton("append", "Append a navbarMenu")
  )
)
server <- function(input, output, session) {
  observeEvent(input$prepend, {
    id <- paste0("Dropdown", input$prepend, "p")
    prependTab(inputId = "tabs",
      navbarMenu(id,
        tabPanel("Drop1", paste("Drop1 page from", id)),
        tabPanel("Drop2", paste("Drop2 page from", id)),
        "------",
        "Header",
        tabPanel("Drop3", paste("Drop3 page from", id))
      )
    )
  })
  observeEvent(input$append, {
    id <- paste0("Dropdown", input$append, "a")
    appendTab(inputId = "tabs",
      navbarMenu(id,
        tabPanel("Drop1", paste("Drop1 page from", id)),
        tabPanel("Drop2", paste("Drop2 page from", id)),
        "------",
        "Header",
        tabPanel("Drop3", paste("Drop3 page from", id))
      )
    )
  })
}
```

```
    shinyApp(ui, server)

}
```

---

insertUI                            *Insert UI objects*

---

### Description

Insert a UI object into the app.

### Usage

```
insertUI(selector, where = c("beforeBegin", "afterBegin", "beforeEnd",
  "afterEnd"), ui, multiple = FALSE, immediate = FALSE,
  session = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| selector | A string that is accepted by jQuery's selector (i.e. the string s to be placed in a $(s) jQuery call). This selector will determine the element(s) relative to which you want to insert your UI object. |
| where | Where your UI object should go relative to the selector: |
| | beforeBegin  Before the selector element itself |
| | afterBegin  Just inside the selector element, before its first child |
| | beforeEnd  Just inside the selector element, after its last child (default) |
| | afterEnd  After the selector element itself |
| | Adapted from here. |
| ui | The UI object you want to insert. This can be anything that you usually put inside your apps's ui function. If you're inserting multiple elements in one call, make sure to wrap them in either a tagList() or a tags$div() (the latter option has the advantage that you can give it an id to make it easier to reference or remove it later on). If you want to insert raw html, use ui = HTML(). |
| multiple | In case your selector matches more than one element, multiple determines whether Shiny should insert the UI object relative to all matched elements or just relative to the first matched element (default). |
| immediate | Whether the UI object should be immediately inserted into the app when you call insertUI, or whether Shiny should wait until all outputs have been updated and all observers have been run (default). |
| session | The shiny session within which to call insertUI. |

**Details**

This function allows you to dynamically add an arbitrarily large UI object into your app, whenever you want, as many times as you want. Unlike renderUI(), the UI generated with insertUI is not updatable as a whole: once it's created, it stays there. Each new call to insertUI creates more UI objects, in addition to the ones already there (all independent from one another). To update a part of the UI (ex: an input object), you must use the appropriate render function or a customized reactive function. To remove any part of your UI, use removeUI().

**See Also**

removeUI()

**Examples**

```
## Only run this example in interactive R sessions
if (interactive()) {
# Define UI
ui <- fluidPage(
  actionButton("add", "Add UI")
)

# Server logic
server <- function(input, output, session) {
  observeEvent(input$add, {
    insertUI(
      selector = "#add",
      where = "afterEnd",
      ui = textInput(paste0("txt", input$add),
                     "Insert some text")
    )
  })
}

# Complete app with UI and server components
shinyApp(ui, server)
}
```

---

installExprFunction          *Install an expression as a function*

---

**Description**

Installs an expression in the given environment as a function, and registers debug hooks so that breakpoints may be set in the function.

## Usage

```
installExprFunction(expr, name, eval.env = parent.frame(2),
  quoted = FALSE, assign.env = parent.frame(1),
  label = deparse(sys.call(-1)[[1]]), wrappedWithLabel = TRUE,
  ..stacktraceon = FALSE)
```

## Arguments

| | |
|---|---|
| expr | A quoted or unquoted expression |
| name | The name the function should be given |
| eval.env | The desired environment for the function. Defaults to the calling environment two steps back. |
| quoted | Is the expression quoted? |
| assign.env | The environment in which the function should be assigned. |
| label | A label for the object to be shown in the debugger. Defaults to the name of the calling function. |
| wrappedWithLabel, ..stacktraceon | |
| | Advanced use only. For stack manipulation purposes; see [stacktrace()](). |

## Details

This function can replace exprToFunction as follows: we may use func <-exprToFunction(expr) if we do not want the debug hooks, or installExprFunction(expr,"func") if we do. Both approaches create a function named func in the current environment.

## See Also

Wraps [exprToFunction()](); see that method's documentation for more documentation and examples.

---

| invalidateLater | *Scheduled Invalidation* |
|---|---|

---

## Description

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

## Usage

```
invalidateLater(millis, session = getDefaultReactiveDomain())
```

## 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. |

**Details**

If this is placed within an observer or reactive expression, that object will be invalidated (and re-execute) after the interval has passed. The re-execution will reset the invalidation flag, so in a typical use case, the object will keep re-executing and waiting for the specified interval. It's possible to stop this cycle by adding conditional logic that prevents the invalidateLater from being run.

**See Also**

[reactiveTimer()](#) is a slightly less safe alternative.

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("n", "Number of observations", 2, 1000, 500),
  plotOutput("plot")
)

server <- 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)
    hist(rnorm(isolate(input$n)))
  })
}

shinyApp(ui, server)
}
```

---

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 [base::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
```

---

| isRunning | *Check whether a Shiny application is running* |
| --- | --- |

---

### Description

This function tests whether a Shiny application is currently running.

### Usage

```
isRunning()
```

## Value

TRUE if a Shiny application is currently running. Otherwise, FALSE.

---

knitr_methods *Knitr S3 methods*

---

## Description

These S3 methods are necessary to help Shiny applications and UI chunks embed themselves in knitr/rmarkdown documents.

## Usage

```
knit_print.shiny.appobj(x, ...)

knit_print.shiny.render.function(x, ..., inline = FALSE)

knit_print.reactive(x, ..., inline = FALSE)
```

## Arguments

| | |
|---|---|
| x | Object to knit_print |
| ... | Additional knit_print arguments |
| inline | Whether the object is printed inline. |

---

loadSupport *Load an app's supporting R files*

---

## Description

Loads all of the supporting R files of a Shiny application. Specifically, this function loads any top-level supporting .R files in the R/ directory adjacent to the app.R/server.R/ui.R files.

## Usage

```
loadSupport(appDir, renv = new.env(parent = globalenv()),
  globalrenv = globalenv())
```

## Arguments

| | |
|---|---|
| appDir | The application directory |
| renv | The environmeny in which the files in the R/ directory should be evaluated. |
| globalrenv | The environment in which global.R should be evaluated. If NULL, global.R will not be evaluated at all. |

## Details

At the moment, this function is "opt-in" and only called if the option `shiny.autoload.r` is set to `TRUE`.

The files are sourced in alphabetical order (as determined by [list.files](#)). `global.R` is evaluated before the supporting R files in the `R/` directory.

---

makeReactiveBinding           *Make a reactive variable*

---

## Description

Turns a normal variable into a reactive variable, that is, one that has reactive semantics when assigned or read in the usual ways. The variable may already exist; if so, its value will be used as the initial value of the reactive variable (or `NULL` if the variable did not exist).

## Usage

```
makeReactiveBinding(symbol, env = parent.frame())
```

## Arguments

symbol          A character string indicating the name of the variable that should be made reactive

env             The environment that will contain the reactive variable

## Value

None.

## Examples

```
## Not run:
a <- 10
makeReactiveBinding("a")
b <- reactive(a * -1)
observe(print(b()))
a <- 20

## End(Not run)
```

---

markRenderFunction          *Mark a function as a render function*

---

### Description

Should be called by implementers of renderXXX functions in order to mark their return values as Shiny render functions, and to provide a hint to Shiny regarding what UI function is most commonly used with this type of render function. This can be used in R Markdown documents to create complete output widgets out of just the render function.

### Usage

```
markRenderFunction(uiFunc, renderFunc, outputArgs = list())
```

### Arguments

uiFunc          A function that renders Shiny UI. Must take a single argument: an output ID.

renderFunc      A function that is suitable for assigning to a Shiny output slot.

outputArgs      A list of arguments to pass to the uiFunc. Render functions should include outputArgs = list() in their own parameter list, and pass through the value to markRenderFunction, to allow app authors to customize outputs. (Currently, this is only supported for dynamically generated UIs, such as those created by Shiny code snippets embedded in R Markdown documents).

### Value

The renderFunc function, with annotations.

---

maskReactiveContext          *Evaluate an expression without a reactive context*

---

### Description

Temporarily blocks the current reactive context and evaluates the given expression. Any attempt to directly access reactive values or expressions in expr will give the same results as doing it at the top-level (by default, an error).

### Usage

```
maskReactiveContext(expr)
```

### Arguments

expr            An expression to evaluate.

**Value**

The value of expr.

**See Also**

[isolate()]

---

memoryCache                          *Create a memory cache object*

---

**Description**

A memory cache object is a key-value store that saves the values in an environment. Objects can be stored and retrieved using the get() and set() methods. Objects are automatically pruned from the cache according to the parameters max_size, max_age, max_n, and evict.

**Usage**

```
memoryCache(max_size = 10 * 1024^2, max_age = Inf, max_n = Inf,
  evict = c("lru", "fifo"), missing = key_missing(),
  exec_missing = FALSE, logfile = NULL)
```

**Arguments**

max_size        Maximum size of the cache, in bytes. If the cache exceeds this size, cached objects will be removed according to the value of the evict. Use Inf for no size limit.

max_age         Maximum age of files in cache before they are evicted, in seconds. Use Inf for no age limit.

max_n           Maximum number of objects in the cache. If the number of objects exceeds this value, then cached objects will be removed according to the value of evict. Use Inf for no limit of number of items.

evict           The eviction policy to use to decide which objects are removed when a cache pruning occurs. Currently, "lru" and "fifo" are supported.

missing         A value to return or a function to execute when get(key) is called but the key is not present in the cache. The default is a [key_missing()] object. If it is a function to execute, the function must take one argument (the key), and you must also use exec_missing = TRUE. If it is a function, it is useful in most cases for it to throw an error, although another option is to return a value. If a value is returned, that value will in turn be returned by get(). See section Missing keys for more information.

exec_missing    If FALSE (the default), then treat missing as a value to return when get() results in a cache miss. If TRUE, treat missing as a function to execute when get() results in a cache miss.

logfile         An optional filename or connection object to where logging information will be written. To log to the console, use stdout().

## Details

In a `MemoryCache`, R objects are stored directly in the cache; they are not *not* serialized before being stored in the cache. This contrasts with other cache types, like [diskCache()](), where objects are serialized, and the serialized object is cached. This can result in some differences of behavior. For example, as long as an object is stored in a MemoryCache, it will not be garbage collected.

## Missing keys

The `missing` and `exec_missing` parameters controls what happens when `get()` is called with a key that is not in the cache (a cache miss). The default behavior is to return a [key_missing()]() object. This is a *sentinel value* that indicates that the key was not present in the cache. You can test if the returned value represents a missing key by using the [is.key_missing()]() function. You can also have `get()` return a different sentinel value, like `NULL`. If you want to throw an error on a cache miss, you can do so by providing a function for `missing` that takes one argument, the key, and also use `exec_missing=TRUE`.

When the cache is created, you can supply a value for `missing`, which sets the default value to be returned for missing values. It can also be overridden when `get()` is called, by supplying a `missing` argument. For example, if you use `cache$get("mykey",missing = NULL)`, it will return `NULL` if the key is not in the cache.

If your cache is configured so that `get()` returns a sentinel value to represent a cache miss, then `set` will also not allow you to store the sentinel value in the cache. It will throw an error if you attempt to do so.

Instead of returning the same sentinel value each time there is cache miss, the cache can execute a function each time `get()` encounters missing key. If the function returns a value, then `get()` will in turn return that value. However, a more common use is for the function to throw an error. If an error is thrown, then `get()` will not return a value.

To do this, pass a one-argument function to `missing`, and use `exec_missing=TRUE`. For example, if you want to throw an error that prints the missing key, you could do this:

```
diskCache(
  missing = function(key) {
    stop("Attempted to get missing key: ", key)
  },
  exec_missing = TRUE
)
```

If you use this, the code that calls `get()` should be wrapped with [tryCatch()]() to gracefully handle missing keys.

## Cache pruning

Cache pruning occurs when `set()` is called, or it can be invoked manually by calling `prune()`.

When a pruning occurs, if there are any objects that are older than `max_age`, they will be removed.

The `max_size` and `max_n` parameters are applied to the cache as a whole, in contrast to `max_age`, which is applied to each object individually.

If the number of objects in the cache exceeds `max_n`, then objects will be removed from the cache according to the eviction policy, which is set with the `evict` parameter. Objects will be removed so that the number of items is `max_n`.

If the size of the objects in the cache exceeds `max_size`, then objects will be removed from the cache. Objects will be removed from the cache so that the total size remains under `max_size`. Note that the size is calculated using the size of the files, not the size of disk space used by the files — these two values can differ because of files are stored in blocks on disk. For example, if the block size is 4096 bytes, then a file that is one byte in size will take 4096 bytes on disk.

Another time that objects can be removed from the cache is when `get()` is called. If the target object is older than `max_age`, it will be removed and the cache will report it as a missing value.

**Eviction policies**

If `max_n` or `max_size` are used, then objects will be removed from the cache according to an eviction policy. The available eviction policies are:

`"lru"` Least Recently Used. The least recently used objects will be removed. This uses the filesystem's atime property. Some filesystems do not support atime, or have a very low atime resolution. The DiskCache will check for atime support, and if the filesystem does not support atime, a warning will be issued and the "fifo" policy will be used instead.

`"fifo"` First-in-first-out. The oldest objects will be removed.

**Methods**

A disk cache object has the following methods:

`get(key, missing, exec_missing)` Returns the value associated with `key`. If the key is not in the cache, then it returns the value specified by `missing` or, `missing` is a function and `exec_missing=TRUE`, then executes `missing`. The function can throw an error or return the value. If either of these parameters are specified here, then they will override the defaults that were set when the DiskCache object was created. See section Missing Keys for more information.

`set(key, value)` Stores the `key`-`value` pair in the cache.

`exists(key)` Returns `TRUE` if the cache contains the key, otherwise `FALSE`.

`size()` Returns the number of items currently in the cache.

`keys()` Returns a character vector of all keys currently in the cache.

`reset()` Clears all objects from the cache.

`destroy()` Clears all objects in the cache, and removes the cache directory from disk.

`prune()` Prunes the cache, using the parameters specified by `max_size`, `max_age`, `max_n`, and `evict`.

---

modalButton                    *Create a button for a modal dialog*

---

### Description

When clicked, a `modalButton` will dismiss the modal dialog.

### Usage

```
modalButton(label, icon = NULL)
```

### Arguments

label          The contents of the button or link–usually a text label, but you could also use
               any other HTML, like an image.

icon           An optional [icon()](#) to appear on the button.

### See Also

[modalDialog()](#) for examples.

---

modalDialog                    *Create a modal dialog UI*

---

### Description

This creates the UI for a modal dialog, using Bootstrap's modal class. Modals are typically used
for showing important messages, or for presenting UI that requires input from the user, such as a
username and password input.

### Usage

```
modalDialog(..., title = NULL, footer = modalButton("Dismiss"),
  size = c("m", "s", "l"), easyClose = FALSE, fade = TRUE)
```

### Arguments

...            UI elements for the body of the modal dialog box.

title          An optional title for the dialog.

footer         UI for footer. Use `NULL` for no footer.

size           One of `"s"` for small, `"m"` (the default) for medium, or `"l"` for large.

easyClose      If `TRUE`, the modal dialog can be dismissed by clicking outside the dialog box,
               or be pressing the Escape key. If `FALSE` (the default), the modal dialog can't be
               dismissed in those ways; instead it must be dismissed by clicking on the dismiss
               button, or from a call to [removeModal()](#) on the server.

fade           If `FALSE`, the modal dialog will have no fade-in animation (it will simply appear
               rather than fade in to view).

## Examples

```
if (interactive()) {
# Display an important message that can be dismissed only by clicking the
# dismiss button.
shinyApp(
  ui = basicPage(
    actionButton("show", "Show modal dialog")
  ),
  server = function(input, output) {
    observeEvent(input$show, {
      showModal(modalDialog(
        title = "Important message",
        "This is an important message!"
      ))
    })
  }
)


# Display a message that can be dismissed by clicking outside the modal dialog,
# or by pressing Esc.
shinyApp(
  ui = basicPage(
    actionButton("show", "Show modal dialog")
  ),
  server = function(input, output) {
    observeEvent(input$show, {
      showModal(modalDialog(
        title = "Somewhat important message",
        "This is a somewhat important message.",
        easyClose = TRUE,
        footer = NULL
      ))
    })
  }
)


# Display a modal that requires valid input before continuing.
shinyApp(
  ui = basicPage(
    actionButton("show", "Show modal dialog"),
    verbatimTextOutput("dataInfo")
  ),

  server = function(input, output) {
    # reactiveValues object for storing current data set.
    vals <- reactiveValues(data = NULL)

    # Return the UI for a modal dialog with data selection input. If 'failed' is
    # TRUE, then display a message that the previous value was invalid.
    dataModal <- function(failed = FALSE) {
```

```
    modalDialog(
      textInput("dataset", "Choose data set",
        placeholder = 'Try "mtcars" or "abc"'
      ),
      span('(Try the name of a valid data object like "mtcars", ',
           'then a name of a non-existent object like "abc")'),
      if (failed)
        div(tags$b("Invalid name of data object", style = "color: red;")),

      footer = tagList(
        modalButton("Cancel"),
        actionButton("ok", "OK")
      )
    )
  }

  # Show modal when button is clicked.
  observeEvent(input$show, {
    showModal(dataModal())
  })

  # When OK button is pressed, attempt to load the data set. If successful,
  # remove the modal. If not show another modal, but this time with a failure
  # message.
  observeEvent(input$ok, {
    # Check that data object exists and is data frame.
    if (!is.null(input$dataset) && nzchar(input$dataset) &&
        exists(input$dataset) && is.data.frame(get(input$dataset))) {
      vals$data <- get(input$dataset)
      removeModal()
    } else {
      showModal(dataModal(failed = TRUE))
    }
  })

  # Display information about selected data
  output$dataInfo <- renderPrint({
    if (is.null(vals$data))
      "No data selected"
    else
      summary(vals$data)
  })
  }
)
}
```

---

navbarPage                      *Create a page with a top level navigation bar*

---

**Description**

Create a page that contains a top level navigation bar that can be used to toggle a set of [tabPanel()](tabPanel())
elements.

**Usage**

```
navbarPage(title, ..., id = NULL, selected = NULL,
  position = c("static-top", "fixed-top", "fixed-bottom"),
  header = NULL, footer = NULL, inverse = FALSE,
  collapsible = FALSE, collapsable, fluid = TRUE, responsive = NULL,
  theme = NULL, windowTitle = title)

navbarMenu(title, ..., menuName = title, icon = NULL)
```

**Arguments**

| | |
|---|---|
| title | The title to display in the navbar |
| ... | [tabPanel()](tabPanel()) elements to include in the page. The navbarMenu function also accepts strings, which will be used as menu section headers. If the string is a set of dashes like "----" a horizontal separator will be displayed in the menu. |
| id | If provided, you can use input$id in your server logic to determine which of the current tabs is active. The value will correspond to the value argument that is passed to [tabPanel()](tabPanel()). |
| selected | The value (or, if none was supplied, the title) of the tab that should be selected by default. If NULL, the first tab will be selected. |
| position | Determines whether the navbar should be displayed at the top of the page with normal scrolling behavior ("static-top"), pinned at the top ("fixed-top"), or pinned at the bottom ("fixed-bottom"). Note that using "fixed-top" or "fixed-bottom" will cause the navbar to overlay your body content, unless you add padding, e.g.: tags$style(type="text/css","body {padding-top: 70px;}") |
| header | Tag or list of tags to display as a common header above all tabPanels. |
| footer | Tag or list of tags to display as a common footer below all tabPanels |
| inverse | TRUE to use a dark background and light text for the navigation bar |
| collapsible | TRUE to automatically collapse the navigation elements into a menu when the width of the browser is less than 940 pixels (useful for viewing on smaller touch-screen device) |
| collapsable | Deprecated; use collapsible instead. |
| fluid | TRUE to use a fluid layout. FALSE to use a fixed layout. |
| responsive | This option is deprecated; it is no longer optional with Bootstrap 3. |
| theme | Alternative Bootstrap stylesheet (normally a css file within the www directory). For example, to use the theme located at www/bootstrap.css you would use theme = "bootstrap.css". |
| windowTitle | The title that should be displayed by the browser window. Useful if title is not a string. |

| menuName | A name that identifies this `navbarMenu`. This is needed if you want to insert/remove or show/hide an entire `navbarMenu`. |
| --- | --- |
| icon | Optional icon to appear on a `navbarMenu` tab. |

## Details

The `navbarMenu` function can be used to create an embedded menu within the navbar that in turns includes additional tabPanels (see example below).

## Value

A UI defintion that can be passed to the [shinyUI](#) function.

## See Also

[tabPanel()](#), [tabsetPanel()](#), [updateNavbarPage()](#), [insertTab()](#), [showTab()](#)

Other layout functions: [fillPage](#), [fixedPage](#), [flowLayout](#), [fluidPage](#), [sidebarLayout](#), [splitLayout](#), [verticalLayout](#)

## Examples

```
navbarPage("App Title",
  tabPanel("Plot"),
  tabPanel("Summary"),
  tabPanel("Table")
)

navbarPage("App Title",
  tabPanel("Plot"),
  navbarMenu("More",
    tabPanel("Summary"),
    "----",
    "Section header",
    tabPanel("Table")
  )
)
```

---

navlistPanel                    *Create a navigation list panel*

---

## Description

Create a navigation list panel that provides a list of links on the left which navigate to a set of tabPanels displayed to the right.

## Usage

```
navlistPanel(..., id = NULL, selected = NULL, well = TRUE,
  fluid = TRUE, widths = c(4, 8))
```

**Arguments**

| | |
|---|---|
| `...` | `tabPanel()` elements to include in the navlist |
| `id` | If provided, you can use input$id in your server logic to determine which of the current navlist items 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 navigation item that should be selected by default. If NULL, the first navigation will be selected. |
| `well` | TRUE to place a well (gray rounded rectangle) around the navigation list. |
| `fluid` | TRUE to use fluid layout; FALSE to use fixed layout. |
| `widths` | Column withs of the navigation list and tabset content areas respectively. |

**Details**

You can include headers within the navlistPanel by including plain text elements in the list. Versions of Shiny before 0.11 supported separators with "——", but as of 0.11, separators were no longer supported. This is because version 0.11 switched to Bootstrap 3, which doesn't support separators.

**See Also**

`tabPanel()`, `updateNavlistPanel()`, `insertTab()`, `showTab()`

**Examples**

```
fluidPage(

  titlePanel("Application Title"),

  navlistPanel(
    "Header",
    tabPanel("First"),
    tabPanel("Second"),
    tabPanel("Third")
  )
)
```

---

nearPoints                    *Find rows of data that are near a click/hover/double-click*

---

**Description**

This function returns rows from a data frame which are near a click, hover, or double-click, when used with `plotOutput()`. The rows will be sorted by their distance to the mouse event.

## Usage

```
nearPoints(df, coordinfo, xvar = NULL, yvar = NULL, panelvar1 = NULL,
  panelvar2 = NULL, threshold = 5, maxpoints = NULL,
  addDist = FALSE, allRows = FALSE)
```

## Arguments

| | |
|---|---|
| df | A data frame from which to select rows. |
| coordinfo | The data from a mouse event, such as `input$plot_click`. |
| xvar | A string with the name of the variable on the x or y axis. This must also be the name of a column in `df`. If absent, then this function will try to infer the variable from the brush (only works for ggplot2). |
| yvar | A string with the name of the variable on the x or y axis. This must also be the name of a column in `df`. If absent, then this function will try to infer the variable from the brush (only works for ggplot2). |
| panelvar1 | Each of these is a string with the name of a panel variable. For example, if with ggplot2, you facet on a variable called `cyl`, then you can use `"cyl"` here. However, specifying the panel variable should not be necessary with ggplot2; Shiny should be able to auto-detect the panel variable. |
| panelvar2 | Each of these is a string with the name of a panel variable. For example, if with ggplot2, you facet on a variable called `cyl`, then you can use `"cyl"` here. However, specifying the panel variable should not be necessary with ggplot2; Shiny should be able to auto-detect the panel variable. |
| threshold | A maxmimum distance to the click point; rows in the data frame where the distance to the click is less than `threshold` will be returned. |
| maxpoints | Maximum number of rows to return. If NULL (the default), return all rows that are within the threshold distance. |
| addDist | If TRUE, add a column named `dist_` that contains the distance from the coordinate to the point, in pixels. When no mouse event has yet occured, the value of `dist_` will be `NA`. |
| allRows | If `FALSE` (the default) return a data frame containing the selected rows. If TRUE, the input data frame will have a new column, `selected_`, which indicates whether the row was inside the selected by the mouse event (TRUE) or not (FALSE). |

## Details

It is also possible for this function to return all rows from the input data frame, but with an additional column `selected_`, which indicates which rows of the input data frame are selected by the brush (TRUE for selected, FALSE for not-selected). This is enabled by setting `allRows=TRUE` option. If this is used, the resulting data frame will not be sorted by distance to the mouse event.

The `xvar`, `yvar`, `panelvar1`, and `panelvar2` arguments specify which columns in the data correspond to the x variable, y variable, and panel variables of the plot. For example, if your plot is `plot(x=cars$speed,y=cars$dist)`, and your click variable is named `"cars_click"`, then you would use `nearPoints(cars,input$cars_brush,"speed","dist")`.

**See Also**

[plotOutput()](plotOutput()) for more examples.

**Examples**

```
## Not run:
# Note that in practice, these examples would need to go in reactives
# or observers.

# This would select all points within 5 pixels of the click
nearPoints(mtcars, input$plot_click)

# Select just the nearest point within 10 pixels of the click
nearPoints(mtcars, input$plot_click, threshold = 10, maxpoints = 1)


## End(Not run)
```

---

NS                                 *Namespaced IDs for inputs/outputs*

---

**Description**

The NS function creates namespaced IDs out of bare IDs, by joining them using ns.sep as the delimiter. It is intended for use in Shiny modules. See [http://shiny.rstudio.com/articles/modules.html](http://shiny.rstudio.com/articles/modules.html).

**Usage**

```
NS(namespace, id = NULL)

ns.sep
```

**Arguments**

namespace       The character vector to use for the namespace. This can have any length, though
                a single element is most common. Length 0 will cause the id to be returned
                without a namespace, and length 2 will be interpreted as multiple namespaces,
                in increasing order of specificity (i.e. starting with the top-level namespace).

id              The id string to be namespaced (optional).

**Format**

An object of class character of length 1.

## Details

Shiny applications use IDs to identify inputs and outputs. These IDs must be unique within an application, as accidentally using the same input/output ID more than once will result in unexpected behavior. The traditional solution for preventing name collisions is *namespaces*; a namespace is to an ID as a directory is to a file. Use the NS function to turn a bare ID into a namespaced one, by combining them with ns.sep in between.

## Value

If id is missing, returns a function that expects an id string as its only argument and returns that id with the namespace prepended.

## See Also

<http://shiny.rstudio.com/articles/modules.html>

---

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,
  width = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| value | Initial value. |
| min | Minimum allowed value |
| max | Maximum allowed value |
| step | Interval to use when stepping between min and max |
| width | The width of the input, e.g. '400px', or '100%'; see validateCssUnit(). |

## Value

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

## Server value

A numeric vector of length 1.

## See Also

[updateNumericInput()](updateNumericInput())

Other input elements: [actionButton](actionButton), [checkboxGroupInput](checkboxGroupInput), [checkboxInput](checkboxInput), [dateInput](dateInput), [dateRangeInput](dateRangeInput), [fileInput](fileInput), [passwordInput](passwordInput), [radioButtons](radioButtons), [selectInput](selectInput), [sliderInput](sliderInput), [submitButton](submitButton), [textAreaInput](textAreaInput), [textInput](textInput), [varSelectInput](varSelectInput)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  numericInput("obs", "Observations:", 10, min = 1, max = 100),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({ input$obs })
}
shinyApp(ui, server)
}
```

---

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,
  domain = getDefaultReactiveDomain(), autoDestroy = TRUE,
  ..stacktraceon = TRUE)
```

## 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. A higher value means higher priority: an observer with a higher priority value will execute before all observers with lower priority values. Positive, negative, and zero values are allowed. |
| domain | See [domains]. |
| autoDestroy | If TRUE (the default), the observer will be automatically destroyed when its domain (if any) ends. |
| ..stacktraceon | Advanced use only. For stack manipulation purposes; see `stacktrace()`. |

## 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.

Starting with Shiny 0.10.0, observers are automatically destroyed by default when the [domain] that owns them ends (e.g. when a Shiny session ends).

## 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.

destroy() Stops the observer from executing ever again, even if it is currently scheduled 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.

setAutoDestroy(autoDestroy) Sets whether this observer should be automatically destroyed when its domain (if any) ends. If autoDestroy is TRUE and the domain already ended, then destroy() is called immediately."

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()
```

observeEvent                *Event handler*

## Description

Respond to "event-like" reactive inputs, values, and expressions.

## Usage

```
observeEvent(eventExpr, handlerExpr, event.env = parent.frame(),
  event.quoted = FALSE, handler.env = parent.frame(),
  handler.quoted = FALSE, label = NULL, suspended = FALSE,
  priority = 0, domain = getDefaultReactiveDomain(),
  autoDestroy = TRUE, ignoreNULL = TRUE, ignoreInit = FALSE,
  once = FALSE)

eventReactive(eventExpr, valueExpr, event.env = parent.frame(),
  event.quoted = FALSE, value.env = parent.frame(),
  value.quoted = FALSE, label = NULL,
  domain = getDefaultReactiveDomain(), ignoreNULL = TRUE,
  ignoreInit = FALSE)
```

## Arguments

eventExpr        A (quoted or unquoted) expression that represents the event; this can be a simple
                 reactive value like input$click, a call to a reactive expression like dataset(),
                 or even a complex expression inside curly braces

handlerExpr      The expression to call whenever eventExpr is invalidated. This should be a
                 side-effect-producing action (the return value will be ignored). It will be exe-
                 cuted within an [isolate()](#) scope.

| event.env | The parent environment for eventExpr. By default, this is the calling environment. |
|---|---|
| event.quoted | Is the eventExpr 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(). |
| handler.env | The parent environment for handlerExpr. By default, this is the calling environment. |
| handler.quoted | Is the handlerExpr 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 or reactive, 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. |
| domain | See [domains](#). |
| autoDestroy | If TRUE (the default), the observer will be automatically destroyed when its domain (if any) ends. |
| ignoreNULL | Whether the action should be triggered (or value calculated, in the case of eventReactive) when the input is NULL. See Details. |
| ignoreInit | If TRUE, then, when this observeEvent is first created/initialized, ignore the handlerExpr (the second argument), whether it is otherwise supposed to run or not. The default is FALSE. See Details. |
| once | Whether this observeEvent should be immediately destroyed after the first time that the code in handlerExpr is run. This pattern is useful when you want to subscribe to a event that should only happen once. |
| valueExpr | The expression that produces the return value of the eventReactive. It will be executed within an [isolate()](#) scope. |
| value.env | The parent environment for valueExpr. By default, this is the calling environment. |
| value.quoted | Is the valueExpr 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(). |

### Details

Shiny's reactive programming framework is primarily designed for calculated values (reactive expressions) and side-effect-causing actions (observers) that respond to *any* of their inputs changing. That's often what is desired in Shiny apps, but not always: sometimes you want to wait for a specific action to be taken from the user, like clicking an [actionButton()](#), before calculating an expression or taking an action. A reactive value or expression that is used to trigger other calculations in this way is called an *event*.

These situations demand a more imperative, "event handling" style of programming that is possible–but not particularly intuitive–using the reactive programming primitives observe() and isolate(). observeEvent and eventReactive provide straightforward APIs for event handling that wrap observe and isolate.

Use observeEvent whenever you want to *perform an action* in response to an event. (Note that "recalculate a value" does not generally count as performing an action–see eventReactive for that.) The first argument is the event you want to respond to, and the second argument is a function that should be called whenever the event occurs.

Use eventReactive to create a *calculated value* that only updates in response to an event. This is just like a normal reactive expression except it ignores all the usual invalidations that come from its reactive dependencies; it only invalidates in response to the given event.

**Value**

observeEvent returns an observer reference class object (see observe()). eventReactive returns a reactive expression object (see reactive()).

**ignoreNULL and ignoreInit**

Both observeEvent and eventReactive take an ignoreNULL parameter that affects behavior when the eventExpr evaluates to NULL (or in the special case of an actionButton(), 0). In these cases, if ignoreNULL is TRUE, then an observeEvent will not execute and an eventReactive will raise a silent validation error. This is useful behavior if you don't want to do the action or calculation when your app first starts, but wait for the user to initiate the action first (like a "Submit" button); whereas ignoreNULL=FALSE is desirable if you want to initially perform the action/calculation and just let the user re-initiate it (like a "Recalculate" button).

Likewise, both observeEvent and eventReactive also take in an ignoreInit argument. By default, both of these will run right when they are created (except if, at that moment, eventExpr evaluates to NULL and ignoreNULL is TRUE). But when responding to a click of an action button, it may often be useful to set ignoreInit to TRUE. For example, if you're setting up an observeEvent for a dynamically created button, then ignoreInit = TRUE will guarantee that the action (in handlerExpr) will only be triggered when the button is actually clicked, instead of also being triggered when it is created/initialized. Similarly, if you're setting up an eventReactive that responds to a dynamically created button used to refresh some data (then returned by that eventReactive), then you should use eventReactive([...],ignoreInit = TRUE) if you want to let the user decide if/when they want to refresh the data (since, depending on the app, this may be a computationally expensive operation).

Even though ignoreNULL and ignoreInit can be used for similar purposes they are independent from one another. Here's the result of combining these:

ignoreNULL = TRUE **and** ignoreInit = FALSE This is the default. This combination means that handlerExpr/ valueExpr will run every time that eventExpr is not NULL. If, at the time of the creation of the observeEvent/eventReactive, eventExpr happens to *not* be NULL, then the code runs.

ignoreNULL = FALSE **and** ignoreInit = FALSE This combination means that handlerExpr/valueExpr will run every time no matter what.

ignoreNULL = FALSE **and** ignoreInit = TRUE  This combination means that handlerExpr/valueExpr
will *not* run when the observeEvent/eventReactive is created (because ignoreInit = TRUE),
but it will run every other time.

ignoreNULL = TRUE **and** ignoreInit = TRUE  This combination means that handlerExpr/valueExpr
will *not* run when the observeEvent/eventReactive is created (because ignoreInit = TRUE).
After that, handlerExpr/valueExpr will run every time that eventExpr is not NULL.

### See Also

[actionButton()](#)

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {

  ## App 1: Sample usage
  shinyApp(
    ui = fluidPage(
      column(4,
        numericInput("x", "Value", 5),
        br(),
        actionButton("button", "Show")
      ),
      column(8, tableOutput("table"))
    ),
    server = function(input, output) {
      # Take an action every time button is pressed;
      # here, we just print a message to the console
      observeEvent(input$button, {
        cat("Showing", input$x, "rows\n")
      })
      # Take a reactive dependency on input$button, but
      # not on any of the stuff inside the function
      df <- eventReactive(input$button, {
        head(cars, input$x)
      })
      output$table <- renderTable({
        df()
      })
    }
  )

  ## App 2: Using `once`
  shinyApp(
    ui = basicPage( actionButton("go", "Go")),
    server = function(input, output, session) {
      observeEvent(input$go, {
        print(paste("This will only be printed once; all",
              "subsequent button clicks won't do anything"))
      }, once = TRUE)
    }
```

```
  )

  ## App 3: Using `ignoreInit` and `once`
  shinyApp(
    ui = basicPage(actionButton("go", "Go")),
    server = function(input, output, session) {
      observeEvent(input$go, {
        insertUI("#go", "afterEnd",
                 actionButton("dynamic", "click to remove"))

        # set up an observer that depends on the dynamic
        # input, so that it doesn't run when the input is
        # created, and only runs once after that (since
        # the side effect is remove the input from the DOM)
        observeEvent(input$dynamic, {
          removeUI("#dynamic")
        }, ignoreInit = TRUE, once = TRUE)
      })
    }
  )
}
```

---

onBookmark                     *Add callbacks for Shiny session bookmarking events*

---

### Description

These functions are for registering callbacks on Shiny session events. They should be called within an application's server function.

- onBookmark registers a function that will be called just before Shiny bookmarks state.

- onBookmarked registers a function that will be called just after Shiny bookmarks state.

- onRestore registers a function that will be called when a session is restored, after the server function executes, but before all other reactives, observers and render functions are run.

- onRestored registers a function that will be called after a session is restored. This is similar to onRestore, but it will be called after all reactives, observers, and render functions run, and after results are sent to the client browser. onRestored callbacks can be useful for sending update messages to the client browser.

### Usage

```
onBookmark(fun, session = getDefaultReactiveDomain())

onBookmarked(fun, session = getDefaultReactiveDomain())

onRestore(fun, session = getDefaultReactiveDomain())

onRestored(fun, session = getDefaultReactiveDomain())
```

## Arguments

| | |
|---|---|
| fun | A callback function which takes one argument. |
| session | A shiny session object. |

## Details

All of these functions return a function which can be called with no arguments to cancel the registration.

The callback function that is passed to these functions should take one argument, typically named "state" (for onBookmark, onRestore, and onRestored) or "url" (for onBookmarked).

For onBookmark, the state object has three relevant fields. The values field is an environment which can be used to save arbitrary values (see examples). If the state is being saved to disk (as opposed to being encoded in a URL), the dir field contains the name of a directory which can be used to store extra files. Finally, the state object has an input field, which is simply the application's input object. It can be read, but not modified.

For onRestore and onRestored, the state object is a list. This list contains input, which is a named list of input values to restore, values, which is an environment containing arbitrary values that were saved in onBookmark, and dir, the name of the directory that the state is being restored from, and which could have been used to save extra files.

For onBookmarked, the callback function receives a string with the bookmark URL. This callback function should be used to display UI in the client browser with the bookmark URL. If no callback function is registered, then Shiny will by default display a modal dialog with the bookmark URL.

## Modules

These callbacks may also be used in Shiny modules. When used this way, the inputs and values will automatically be namespaced for the module, and the callback functions registered for the module will only be able to see the module's inputs and values.

## See Also

enableBookmarking for general information on bookmarking.

## Examples

```
## Only run these examples in interactive sessions
if (interactive()) {

# Basic use of onBookmark and onRestore: This app saves the time in its
# arbitrary values, and restores that time when the app is restored.
ui <- function(req) {
  fluidPage(
    textInput("txt", "Input text"),
    bookmarkButton()
  )
}
server <- function(input, output) {
  onBookmark(function(state) {
```

```
    savedTime <- as.character(Sys.time())
    cat("Last saved at", savedTime, "\n")
    # state is a mutable reference object, and we can add arbitrary values to
    # it.
    state$values$time <- savedTime
  })

  onRestore(function(state) {
    cat("Restoring from state bookmarked at", state$values$time, "\n")
  })
}
enableBookmarking("url")
shinyApp(ui, server)



ui <- function(req) {
  fluidPage(
    textInput("txt", "Input text"),
    bookmarkButton()
  )
}
server <- function(input, output, session) {
  lastUpdateTime <- NULL

  observeEvent(input$txt, {
    updateTextInput(session, "txt",
      label = paste0("Input text (Changed ", as.character(Sys.time()), ")")
    )
  })

  onBookmark(function(state) {
    # Save content to a file
    messageFile <- file.path(state$dir, "message.txt")
    cat(as.character(Sys.time()), file = messageFile)
  })

  onRestored(function(state) {
    # Read the file
    messageFile <- file.path(state$dir, "message.txt")
    timeText <- readChar(messageFile, 1000)

    # updateTextInput must be called in onRestored, as opposed to onRestore,
    # because onRestored happens after the client browser is ready.
    updateTextInput(session, "txt",
      label = paste0("Input text (Changed ", timeText, ")")
    )
  })
}
# "server" bookmarking is needed for writing to disk.
enableBookmarking("server")
shinyApp(ui, server)
```

```
# This app has a module, and both the module and the main app code have
# onBookmark and onRestore functions which write and read state$values$hash. The
# module's version of state$values$hash does not conflict with the app's version
# of state$values$hash.
#
# A basic module that captializes text.
capitalizerUI <- function(id) {
  ns <- NS(id)
  wellPanel(
    h4("Text captializer module"),
    textInput(ns("text"), "Enter text:"),
    verbatimTextOutput(ns("out"))
  )
}
capitalizerServer <- function(input, output, session) {
  output$out <- renderText({
    toupper(input$text)
  })
  onBookmark(function(state) {
    state$values$hash <- digest::digest(input$text, "md5")
  })
  onRestore(function(state) {
    if (identical(digest::digest(input$text, "md5"), state$values$hash)) {
      message("Module's input text matches hash ", state$values$hash)
    } else {
      message("Module's input text does not match hash ", state$values$hash)
    }
  })
}
# Main app code
ui <- function(request) {
  fluidPage(
    sidebarLayout(
      sidebarPanel(
        capitalizerUI("tc"),
        textInput("text", "Enter text (not in module):"),
        bookmarkButton()
      ),
      mainPanel()
    )
  )
}
server <- function(input, output, session) {
  callModule(capitalizerServer, "tc")
  onBookmark(function(state) {
    state$values$hash <- digest::digest(input$text, "md5")
  })
  onRestore(function(state) {
    if (identical(digest::digest(input$text, "md5"), state$values$hash)) {
      message("App's input text matches hash ", state$values$hash)
    } else {
      message("App's input text does not match hash ", state$values$hash)
```

```
    }
  })
}
enableBookmarking(store = "url")
shinyApp(ui, server)
}
```

---

onFlush                               *Add callbacks for Shiny session events*

---

### Description

These functions are for registering callbacks on Shiny session events. onFlush registers a function that will be called before Shiny flushes the reactive system. onFlushed registers a function that will be called after Shiny flushes the reactive system. onSessionEnded registers a function to be called after the client has disconnected.

### Usage

```
onFlush(fun, once = TRUE, session = getDefaultReactiveDomain())

onFlushed(fun, once = TRUE, session = getDefaultReactiveDomain())

onSessionEnded(fun, session = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| fun | A callback function. |
| once | Should the function be run once, and then cleared, or should it re-run each time the event occurs. (Only for onFlush and onFlushed.) |
| session | A shiny session object. |

### Details

These functions should be called within the application's server function.

All of these functions return a function which can be called with no arguments to cancel the registration.

### See Also

[onStop()](#) for registering callbacks that will be invoked when the application exits, or when a session ends.

---

onStop                    *Run code after an application or session ends*

---

### Description

This function registers callback functions that are invoked when the application exits (when [runApp()](#)
exits), or after each user session ends (when a client disconnects).

### Usage

```
onStop(fun, session = getDefaultReactiveDomain())
```

### Arguments

fun             A function that will be called after the app has finished running.

session         A scope for when the callback will run. If onStop is called from within the
                server function, this will default to the current session, and the callback will
                be invoked when the current session ends. If onStop is called outside a server
                function, then the callback will be invoked with the application exits. If NULL,
                it is the same as calling onStop outside of the server function, and the callback
                will be invoked when the application exits.

### Value

A function which, if invoked, will cancel the callback.

### See Also

[onSessionEnded()](#) for the same functionality, but at the session level only.

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # Open this application in multiple browsers, then close the browsers.
  shinyApp(
    ui = basicPage("onStop demo"),

    server = function(input, output, session) {
      onStop(function() cat("Session stopped\n"))
    },

    onStart = function() {
      cat("Doing application setup\n")

      onStop(function() {
        cat("Doing application cleanup\n")
      })
    }
```

```
  )
}
# In the example above, onStop() is called inside of onStart(). This is
# the pattern that should be used when creating a shinyApp() object from
# a function, or at the console. If instead you are writing an app.R which
# will be invoked with runApp(), you can do it that way, or put the onStop()
# before the shinyApp() call, as shown below.

## Not run:
# ==== app.R ====
cat("Doing application setup\n")
onStop(function() {
  cat("Doing application cleanup\n")
})

shinyApp(
  ui = basicPage("onStop demo"),

  server = function(input, output, session) {
    onStop(function() cat("Session stopped\n"))
  }
)
# ==== end app.R ====


# Similarly, if you have a global.R, you can call onStop() from there.
# ==== global.R ====
cat("Doing application setup\n")
onStop(function() {
  cat("Doing application cleanup\n")
})
# ==== end global.R ====

## End(Not run)
```

---

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)
```

---

parseQueryString            *Parse a GET query string from a URL*

---

## Description

Returns a named list of key-value pairs.

## Usage

```
parseQueryString(str, nested = FALSE)
```

## Arguments

| | |
|---|---|
| str | The query string. It can have a leading "?" or not. |
| nested | Whether to parse the query string of as a nested list when it contains pairs of square brackets []. For example, the query 'a[i1][j1]=x&b[i1][j1]=y&b[i2][j1]=z' will be parsed as list(a = list(i1 = list(j1 = 'x')),b = list(i1 = list(j1 = 'y'),i2 = list(j1 = 'z'))) when nested = TRUE, and list(`a[i1][j1]` = 'x',`b[i1][j1]` = 'y',`b[i2][j1]` = 'z') when nested = FALSE. |

## Examples

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

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

  output$queryText <- renderText({
    query <- parseQueryString(session$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)
```

---

passwordInput                   *Create a password input control*

---

### Description

Create an password control for entry of passwords.

### Usage

```
passwordInput(inputId, label, value = "", width = NULL,
  placeholder = NULL)
```

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| value | Initial value. |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| placeholder | A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option. |

## Value

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

## Server value

A character string of the password input. The default value is `""` unless value is provided.

## See Also

[updateTextInput()](#)

Other input elements: [actionButton](#), [checkboxGroupInput](#), [checkboxInput](#), [dateInput](#), [dateRangeInput](#), [fileInput](#), [numericInput](#), [radioButtons](#), [selectInput](#), [sliderInput](#), [submitButton](#), [textAreaInput](#), [textInput](#), [varSelectInput](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  passwordInput("password", "Password:"),
  actionButton("go", "Go"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({
    req(input$go)
    isolate(input$password)
  })
}
shinyApp(ui, server)
}
```

---

plotOutput                    *Create an plot or image output element*

---

## Description

Render a [renderPlot()](#) or [renderImage()](#) within an application page.

## Usage

```
imageOutput(outputId, width = "100%", height = "400px",
  click = NULL, dblclick = NULL, hover = NULL, hoverDelay = NULL,
  hoverDelayType = NULL, brush = NULL, clickId = NULL,
  hoverId = NULL, inline = FALSE)

plotOutput(outputId, width = "100%", height = "400px", click = NULL,
```

```
dblclick = NULL, hover = NULL, hoverDelay = NULL,
hoverDelayType = NULL, brush = NULL, clickId = NULL,
hoverId = NULL, inline = FALSE)
```

## Arguments

| | |
|---|---|
| outputId | output variable to read the plot/image from. |
| width, height | Image width/height. Must be a valid CSS unit (like "100%", "400px", "auto") or a number, which will be coerced to a string and have "px" appended. These two arguments are ignored when inline = TRUE, in which case the width/height of a plot must be specified in renderPlot(). Note that, for height, using "auto" or "100%" generally will not work as expected, because of how height is computed with HTML/CSS. |
| click | This can be NULL (the default), a string, or an object created by the [clickOpts()](#) function. If you use a value like "plot_click" (or equivalently, clickOpts(id="plot_click")), the plot will send coordinates to the server whenever it is clicked, and the value will be accessible via input$plot_click. The value will be a named list with x and y elements indicating the mouse position. |
| dblclick | This is just like the click argument, but for double-click events. |
| hover | Similar to the click argument, this can be NULL (the default), a string, or an object created by the [hoverOpts()](#) function. If you use a value like "plot_hover" (or equivalently, hoverOpts(id="plot_hover")), the plot will send coordinates to the server pauses on the plot, and the value will be accessible via input$plot_hover. The value will be a named list with x and y elements indicating the mouse position. To control the hover time or hover delay type, you must use [hoverOpts()](#). |
| hoverDelay | Deprecated; use hover instead. Also see the [hoverOpts()](#) function. |
| hoverDelayType | Deprecated; use hover instead. Also see the [hoverOpts()](#) function. |
| brush | Similar to the click argument, this can be NULL (the default), a string, or an object created by the [brushOpts()](#) function. If you use a value like "plot_brush" (or equivalently, brushOpts(id="plot_brush")), the plot will allow the user to "brush" in the plotting area, and will send information about the brushed area to the server, and the value will be accessible via input$plot_brush. Brushing means that the user will be able to draw a rectangle in the plotting area and drag it around. The value will be a named list with xmin, xmax, ymin, and ymax elements indicating the brush area. To control the brush behavior, use [brushOpts()](#). Multiple imageOutput/plotOutput calls may share the same id value; brushing one image or plot will cause any other brushes with the same id to disappear. |
| clickId | Deprecated; use click instead. Also see the [clickOpts()](#) function. |
| hoverId | Deprecated; use hover instead. Also see the [hoverOpts()](#) function. |
| inline | use an inline (span()) or block container (div()) for the output |

## Value

A plot or image output element that can be included in a panel.

**Interactive plots**

Plots and images in Shiny support mouse-based interaction, via clicking, double-clicking, hovering, and brushing. When these interaction events occur, the mouse coordinates will be sent to the server as input$ variables, as specified by click, dblclick, hover, or brush.

For plotOutput, the coordinates will be sent scaled to the data space, if possible. (At the moment, plots generated by base graphics and ggplot2 support this scaling, although plots generated by lattice and others do not.) If scaling is not possible, the raw pixel coordinates will be sent. For imageOutput, the coordinates will be sent in raw pixel coordinates.

With ggplot2 graphics, the code in renderPlot should return a ggplot object; if instead the code prints the ggplot2 object with something like print(p), then the coordinates for interactive graphics will not be properly scaled to the data space.

**Note**

The arguments clickId and hoverId only work for R base graphics (see the **graphics** package). They do not work for **grid**-based graphics, such as **ggplot2**, **lattice**, and so on.

**See Also**

For the corresponding server-side functions, see renderPlot() and renderImage().

**Examples**

```
# Only run these examples in interactive R sessions
if (interactive()) {

# A basic shiny app with a plotOutput
shinyApp(
  ui = fluidPage(
    sidebarLayout(
      sidebarPanel(
        actionButton("newplot", "New plot")
      ),
      mainPanel(
        plotOutput("plot")
      )
    )
  ),
  server = function(input, output) {
    output$plot <- renderPlot({
      input$newplot
      # Add a little noise to the cars data
      cars2 <- cars + rnorm(nrow(cars))
      plot(cars2)
    })
  }
)


# A demonstration of clicking, hovering, and brushing
```

```
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4,
        plotOutput("plot", height=300,
          click = "plot_click",  # Equiv, to click=clickOpts(id="plot_click")
          hover = hoverOpts(id = "plot_hover", delayType = "throttle"),
          brush = brushOpts(id = "plot_brush")
        ),
        h4("Clicked points"),
        tableOutput("plot_clickedpoints"),
        h4("Brushed points"),
        tableOutput("plot_brushedpoints")
      ),
      column(width = 4,
        verbatimTextOutput("plot_clickinfo"),
        verbatimTextOutput("plot_hoverinfo")
      ),
      column(width = 4,
        wellPanel(actionButton("newplot", "New plot")),
        verbatimTextOutput("plot_brushinfo")
      )
    )
  ),
  server = function(input, output, session) {
    data <- reactive({
      input$newplot
      # Add a little noise to the cars data so the points move
      cars + rnorm(nrow(cars))
    })
    output$plot <- renderPlot({
      d <- data()
      plot(d$speed, d$dist)
    })
    output$plot_clickinfo <- renderPrint({
      cat("Click:\n")
      str(input$plot_click)
    })
    output$plot_hoverinfo <- renderPrint({
      cat("Hover (throttled):\n")
      str(input$plot_hover)
    })
    output$plot_brushinfo <- renderPrint({
      cat("Brush (debounced):\n")
      str(input$plot_brush)
    })
    output$plot_clickedpoints <- renderTable({
      # For base graphics, we need to specify columns, though for ggplot2,
      # it's usually not necessary.
      res <- nearPoints(data(), input$plot_click, "speed", "dist")
      if (nrow(res) == 0)
        return()
      res
```

```
    })
    output$plot_brushedpoints <- renderTable({
      res <- brushedPoints(data(), input$plot_brush, "speed", "dist")
      if (nrow(res) == 0)
        return()
      res
    })
  }
)


# Demo of clicking, hovering, brushing with imageOutput
# Note that coordinates are in pixels
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4,
        imageOutput("image", height=300,
          click = "image_click",
          hover = hoverOpts(
            id = "image_hover",
            delay = 500,
            delayType = "throttle"
          ),
          brush = brushOpts(id = "image_brush")
        )
      ),
      column(width = 4,
        verbatimTextOutput("image_clickinfo"),
        verbatimTextOutput("image_hoverinfo")
      ),
      column(width = 4,
        wellPanel(actionButton("newimage", "New image")),
        verbatimTextOutput("image_brushinfo")
      )
    )
  ),
  server = function(input, output, session) {
    output$image <- renderImage({
      input$newimage

      # Get width and height of image output
      width  <- session$clientData$output_image_width
      height <- session$clientData$output_image_height

      # Write to a temporary PNG file
      outfile <- tempfile(fileext = ".png")

      png(outfile, width=width, height=height)
      plot(rnorm(200), rnorm(200))
      dev.off()

      # Return a list containing information about the image
```

```
      list(
        src = outfile,
        contentType = "image/png",
        width = width,
        height = height,
        alt = "This is alternate text"
      )
    })
    output$image_clickinfo <- renderPrint({
      cat("Click:\n")
      str(input$image_click)
    })
    output$image_hoverinfo <- renderPrint({
      cat("Hover (throttled):\n")
      str(input$image_hover)
    })
    output$image_brushinfo <- renderPrint({
      cat("Brush (debounced):\n")
      str(input$image_brush)
    })
  }
)

  }
```

---

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 grDevices::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 grDevices::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 `grDevices::png()`), then `Cairo::CairoPNG()`, and finally `grDevices::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).

---

| Progress | *Reporting progress (object-oriented API)* |
|---|---|

---

## Description

Reports progress to the user during long-running operations.

## Arguments

| | |
|---|---|
| session | The Shiny session object, as provided by shinyServer to the server function. |
| min | The value that represents the starting point of the progress bar. Must be less than max. |
| max | The value that represents the end of the progress bar. Must be greater than min. |
| message | A single-element character vector; the message to be displayed to the user, or NULL to hide the current message (if any). |
| detail | A single-element character vector; the detail message to be displayed to the user, or NULL to hide the current detail message (if any). The detail message will be shown with a de-emphasized appearance relative to message. |
| value | A numeric value at which to set the progress bar, relative to min and max. |
| style | Progress display style. If "notification" (the default), the progress indicator will show using Shiny's notification API. If "old", use the same HTML and CSS used in Shiny 0.13.2 and below (this is for backward-compatibility). |
| amount | Single-element numeric vector; the value at which to set the progress bar, relative to min and max. NULL hides the progress bar, if it is currently visible. |
| amount | For the inc() method, a numeric value to increment the progress bar. |

## Details

This package exposes two distinct programming APIs for working with progress. `withProgress()` and `setProgress()` together provide a simple function-based interface, while the `Progress` reference class provides an object-oriented API.

Instantiating a `Progress` object causes a progress panel to be created, and it will be displayed the first time the set method is called. Calling close will cause the progress panel to be removed.

As of version 0.14, the progress indicators use Shiny's new notification API. If you want to use the old styling (for example, you may have used customized CSS), you can use style="old" each time

you call Progress$new(). If you don't want to set the style each time Progress$new is called, you can instead call shinyOptions(progress.style="old") just once, inside the server function.

**Methods**

initialize(session, min = 0, max = 1)  Creates a new progress panel (but does not display it).

set(value = NULL, message = NULL, detail = NULL)  Updates the progress panel.  When called the first time, the progress panel is displayed.

inc(amount = 0.1, message = NULL, detail = NULL)  Like set, this updates the progress panel. The difference is that inc increases the progress bar by amount, instead of setting it to a specific value.

close()  Removes the progress panel. Future calls to set and close will be ignored.

**See Also**

withProgress()

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  plotOutput("plot")
)

server <- function(input, output, session) {
  output$plot <- renderPlot({
    progress <- Progress$new(session, min=1, max=15)
    on.exit(progress$close())

    progress$set(message = 'Calculation in progress',
                 detail = 'This may take a while...')

    for (i in 1:15) {
      progress$set(value = i)
      Sys.sleep(0.5)
    }
    plot(cars)
  })
}

shinyApp(ui, server)
}
```

---

radioButtons                    *Create radio buttons*

---

## Description

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

## Usage

```
radioButtons(inputId, label, choices = NULL, selected = NULL,
  inline = FALSE, width = NULL, choiceNames = NULL,
  choiceValues = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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). If this argument is provided, then choiceNames and choiceValues must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings. |
| selected | The initially selected value (if not specified then defaults to the first value) |
| inline | If TRUE, render the choices inline (i.e. horizontally) |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| choiceNames, choiceValues | |
| | List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, choiceNames and choiceValues must have the same length). If either of these arguments is provided, then the other *must* be provided and choices *must not* be provided. The advantage of using both of these over a named list for choices is that choiceNames allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples. |

## Details

If you need to represent a "None selected" state, it's possible to default the radio buttons to have no options selected by using selected = character(0). However, this is not recommended, as it gives the user no way to return to that state once they've made a selection. Instead, consider having the first of your choices be c("None selected" = "").

## Value

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

**Server value**

A character string containing the value of the selected button.

**See Also**

[updateRadioButtons()](#)

Other input elements: [actionButton](#), [checkboxGroupInput](#), [checkboxInput](#), [dateInput](#), [dateRangeInput](#), [fileInput](#), [numericInput](#), [passwordInput](#), [selectInput](#), [sliderInput](#), [submitButton](#), [textAreaInput](#), [textInput](#), [varSelectInput](#)

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  radioButtons("dist", "Distribution type:",
               c("Normal" = "norm",
                 "Uniform" = "unif",
                 "Log-normal" = "lnorm",
                 "Exponential" = "exp")),
  plotOutput("distPlot")
)

server <- function(input, output) {
  output$distPlot <- renderPlot({
    dist <- switch(input$dist,
                   norm = rnorm,
                   unif = runif,
                   lnorm = rlnorm,
                   exp = rexp,
                   rnorm)

    hist(dist(500))
  })
}

shinyApp(ui, server)

ui <- fluidPage(
  radioButtons("rb", "Choose one:",
               choiceNames = list(
                 icon("calendar"),
                 HTML("<p style='color:red;'>Red Text</p>"),
                 "Normal text"
               ),
               choiceValues = list(
                 "icon", "html", "text"
               )),
  textOutput("txt")
)
```

```
server <- function(input, output) {
  output$txt <- renderText({
    paste("You chose", input$rb)
  })
}

shinyApp(ui, server)
}
```

---

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,
  domain = getDefaultReactiveDomain(), ..stacktraceon = TRUE)

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. |
| domain | See [domains](#). |
| ..stacktraceon | Advanced use only. For stack manipulation purposes; see [stacktrace()](#). |

### 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()](reactivePoll())

## Examples

```
## Not run:
# Per-session reactive file reader
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, NULL, 'data.csv', read.csv)
function(input, output, session) {
  output$data <- renderTable({
    fileData()
  })
}

## End(Not run)
```

---

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()](reactiveFileReader())

## Examples

```
function(input, output, session) {

  data <- reactivePoll(1000, session,
    # This function returns the time that log_file was last modified
    checkFunc = function() {
```

```
      if (file.exists(log_file))
        file.info(log_file)$mtime[1]
      else
        ""
    },
    # This function returns the content of log_file
    valueFunc = function() {
      read.csv(log_file)
    }
  )

  output$dataTable <- renderTable({
    data()
  })
}
```

---

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 = getDefaultReactiveDomain())
```

### 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 [base::Sys.time()](#)).

**See Also**

[invalidateLater()](invalidateLater())

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("n", "Number of observations", 2, 1000, 500),
  plotOutput("plot")
)

server <- function(input, output) {

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

  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(rnorm(isolate(input$n)))
  })
}

shinyApp(ui, server)
}
```

---

reactiveVal                    *Create a (single) reactive value*

---

**Description**

The reactiveVal function is used to construct a "reactive value" object. This is an object used for reading and writing a value, like a variable, but with special capabilities for reactive programming. When you read the value out of a reactiveVal object, the calling reactive expression takes a dependency, and when you change the value, it notifies any reactives that previously depended on that value.

## Usage

```
reactiveVal(value = NULL, label = NULL)
```

## Arguments

value            An optional initial value.

label            An optional label, for debugging purposes (see `reactlog()`). If missing, a label will be automatically created.

## Details

reactiveVal is very similar to `reactiveValues()`, except that the former is for a single reactive value (like a variable), whereas the latter lets you conveniently use multiple reactive values by name (like a named list of variables). For a one-off reactive value, it's more natural to use `reactiveVal`. See the Examples section for an illustration.

## Value

A function. Call the function with no arguments to (reactively) read the value; call the function with a single argument to set the value.

## Examples

```
## Not run:

# Create the object by calling reactiveVal
r <- reactiveVal()

# Set the value by calling with an argument
r(10)

# Read the value by calling without arguments
r()


## End(Not run)

## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  actionButton("minus", "-1"),
  actionButton("plus", "+1"),
  br(),
  textOutput("value")
)

# The comments below show the equivalent logic using reactiveValues()
server <- function(input, output, session) {
  value <- reactiveVal(0)       # rv <- reactiveValues(value = 0)
```

```
  observeEvent(input$minus, {
    newValue <- value() - 1      # newValue <- rv$value - 1
    value(newValue)              # rv$value <- newValue
  })

  observeEvent(input$plus, {
    newValue <- value() + 1      # newValue <- rv$value + 1
    value(newValue)              # rv$value <- newValue
  })

  output$value <- renderText({
    value()                      # rv$value
  })
}

shinyApp(ui, server)

}
```

| 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. Note that values taken from the reactiveValues object are reactive, but the reactiveValues object itself is not.

### 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 want or expect [base::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))
```

---

reactlog *Reactive Log Visualizer*

---

### Description

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

### Usage

```
reactlog()

reactlogShow(time = TRUE)

showReactLog(time = TRUE)

reactlogReset()
```

### Arguments

time          A boolean that specifies whether or not to display the time that each reactive
              takes to calculate a result.

### 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 reactlogShow 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 reactlogShow() 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.

## Functions

- reactlog: Return a list of reactive information. Can be used in conjunction with [react-log::reactlog_show](#) to later display the reactlog graph.
- reactlogShow: Display a full reactlog graph for all sessions.
- showReactLog: This function is deprecated. You should use [reactlogShow()](#)
- reactlogReset: Resets the entire reactlog stack. Useful for debugging and removing all prior reactive history.

---

registerInputHandler      *Register an Input Handler*

---

## Description

Adds an input handler for data of this type. When called, Shiny will use the function provided to refine the data passed back from the client (after being deserialized by jsonlite) before making it available in the input variable of the server.R file.

## Usage

```
registerInputHandler(type, fun, force = FALSE)
```

## Arguments

| | |
|---|---|
| type | The type for which the handler should be added — should be a single-element character vector. |
| fun | The handler function. This is the function that will be used to parse the data delivered from the client before it is available in the input variable. The function will be called with the following three parameters: |

    1. The value of this input as provided by the client, deserialized using jsonlite.

    2. The shinysession in which the input exists.

    3. The name of the input.

| | |
|---|---|
| force | If TRUE, will overwrite any existing handler without warning. If FALSE, will throw an error if this class already has a handler defined. |

**Details**

This function will register the handler for the duration of the R process (unless Shiny is explicitly reloaded). For that reason, the `type` used should be very specific to this package to minimize the risk of colliding with another Shiny package which might use this data type name. We recommend the format of "packageName.widgetName".

Currently Shiny registers the following handlers: `shiny.matrix`, `shiny.number`, and `shiny.date`.

The `type` of a custom Shiny Input widget will be deduced using the `getType()` JavaScript function on the registered Shiny inputBinding.

**See Also**

[removeInputHandler()](#)

**Examples**

```
## Not run:
# Register an input handler which rounds a input number to the nearest integer
registerInputHandler("mypackage.validint", function(x, shinysession, name) {
  if (is.null(x)) return(NA)
  round(x)
})

## On the Javascript side, the associated input binding must have a corresponding getType method:
getType: function(el) {
  return "mypackage.validint";
}


## End(Not run)
```

---

removeInputHandler            *Deregister an Input Handler*

---

**Description**

Removes an Input Handler. Rather than using the previously specified handler for data of this type, the default jsonlite serialization will be used.

**Usage**

```
removeInputHandler(type)
```

**Arguments**

type            The type for which handlers should be removed.

## Value

The handler previously associated with this type, if one existed. Otherwise, NULL.

## See Also

[registerInputHandler()](#)

---

removeUI                     *Remove UI objects*

---

## Description

Remove a UI object from the app.

## Usage

```
removeUI(selector, multiple = FALSE, immediate = FALSE,
  session = getDefaultReactiveDomain())
```

## Arguments

selector
A string that is accepted by jQuery's selector (i.e. the string s to be placed in a $(s) jQuery call). This selector will determine the element(s) to be removed. If you want to remove a Shiny input or output, note that many of these are wrapped in divs, so you may need to use a somewhat complex selector — see the Examples below. (Alternatively, you could also wrap the inputs/outputs that you want to be able to remove easily in a div with an id.)

multiple
In case your selector matches more than one element, multiple determines whether Shiny should remove all the matched elements or just the first matched element (default).

immediate
Whether the element(s) should be immediately removed from the app when you call removeUI, or whether Shiny should wait until all outputs have been updated and all observers have been run (default).

session
The shiny session within which to call removeUI.

## Details

This function allows you to remove any part of your UI. Once removeUI is executed on some element, it is gone forever.

While it may be a particularly useful pattern to pair this with [insertUI()](#) (to remove some UI you had previously inserted), there is no restriction on what you can use removeUI on. Any element that can be selected through a jQuery selector can be removed through this function.

## See Also

[insertUI()](#)

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
# Define UI
ui <- fluidPage(
  actionButton("rmv", "Remove UI"),
  textInput("txt", "This is no longer useful")
)

# Server logic
server <- function(input, output, session) {
  observeEvent(input$rmv, {
    removeUI(
      selector = "div:has(> #txt)"
    )
  })
}

# Complete app with UI and server components
shinyApp(ui, server)
}
```

---

renderCachedPlot                *Plot output with cached images*

---

## Description

Renders a reactive plot, with plot images cached to disk.

## Usage

```
renderCachedPlot(expr, cacheKeyExpr, sizePolicy = sizeGrowthRatio(width =
  400, height = 400, growthRate = 1.2), res = 72, cache = "app", ...,
  outputArgs = list())
```

## Arguments

| | |
|---|---|
| expr | An expression that generates a plot. |
| cacheKeyExpr | An expression that returns a cache key. This key should be a unique identifier for a plot: the assumption is that if the cache key is the same, then the plot will be the same. |
| sizePolicy | A function that takes two arguments, width and height, and returns a list with width and height. The purpose is to round the actual pixel dimensions from the browser to some other dimensions, so that this will not generate and cache images of every possible pixel dimension. See sizeGrowthRatio() for more information on the default sizing policy. |
| res | The resolution of the PNG, in pixels per inch. |

| | |
|---|---|
| cache | The scope of the cache, or a cache object. This can be "app" (the default), "session", or a cache object like a diskCache(). See the Cache Scoping section for more information. |
| ... | Arguments to be passed through to grDevices::png(). These can be used to set the width, height, background color, etc. |
| outputArgs | A list of arguments to be passed through to the implicit call to plotOutput() when renderPlot is used in an interactive R Markdown document. |

## Details

expr is an expression that generates a plot, similar to that in renderPlot. Unlike with renderPlot, this expression does not take reactive dependencies. It is re-executed only when the cache key changes.

cacheKeyExpr is an expression which, when evaluated, returns an object which will be serialized and hashed using the digest::digest() function to generate a string that will be used as a cache key. This key is used to identify the contents of the plot: if the cache key is the same as a previous time, it assumes that the plot is the same and can be retrieved from the cache.

This cacheKeyExpr is reactive, and so it will be re-evaluated when any upstream reactives are invalidated. This will also trigger re-execution of the plotting expression, expr.

The key should consist of "normal" R objects, like vectors and lists. Lists should in turn contain other normal R objects. If the key contains environments, external pointers, or reference objects — or even if it has such objects attached as attributes — then it is possible that it will change unpredictably even when you do not expect it to. Additionally, because the entire key is serialized and hashed, if it contains a very large object — a large data set, for example — there may be a noticeable performance penalty.

If you face these issues with the cache key, you can work around them by extracting out the important parts of the objects, and/or by converting them to normal R objects before returning them. Your expression could even serialize and hash that information in an efficient way and return a string, which will in turn be hashed (very quickly) by the digest::digest() function.

Internally, the result from cacheKeyExpr is combined with the name of the output (if you assign it to output$plot1, it will be combined with "plot1") to form the actual key that is used. As a result, even if there are multiple plots that have the same cacheKeyExpr, they will not have cache key collisions.

## Cache scoping

There are a number of different ways you may want to scope the cache. For example, you may want each user session to have their own plot cache, or you may want each run of the application to have a cache (shared among possibly multiple simultaneous user sessions), or you may want to have a cache that persists even after the application is shut down and started again.

To control the scope of the cache, use the cache parameter. There are two ways of having Shiny automatically create and clean up the disk cache.

1 To scope the cache to one run of a Shiny application (shared among possibly multiple user sessions), use cache="app". This is the default. The cache will be shared across multiple sessions, so there is potentially a large performance benefit if there are many users of the application. When the application stops running, the cache will be deleted. If plots cannot be safely shared across users, this should not be used.

**2** To scope the cache to one session, use cache="session". When a new user session starts — in other words, when a web browser visits the Shiny application — a new cache will be created on disk for that session. When the session ends, the cache will be deleted. The cache will not be shared across multiple sessions.

If either "app" or "session" is used, the cache will be 10 MB in size, and will be stored stored in memory, using a memoryCache() object. Note that the cache space will be shared among all cached plots within a single application or session.

In some cases, you may want more control over the caching behavior. For example, you may want to use a larger or smaller cache, share a cache among multiple R processes, or you may want the cache to persist across multiple runs of an application, or even across multiple R processes.

To use different settings for an application-scoped cache, you can call shinyOptions() at the top of your app.R, server.R, or global.R. For example, this will create a cache with 20 MB of space instead of the default 10 MB:

```
shinyOptions(cache = memoryCache(size = 20e6))
```

To use different settings for a session-scoped cache, you can call shinyOptions() at the top of your server function. To use the session-scoped cache, you must also call renderCachedPlot with cache="session". This will create a 20 MB cache for the session:

```
function(input, output, session) {
  shinyOptions(cache = memoryCache(size = 20e6))

  output$plot <- renderCachedPlot(
    ...,
    cache = "session"
  )
}
```

If you want to create a cache that is shared across multiple concurrent R processes, you can use a diskCache(). You can create an application-level shared cache by putting this at the top of your app.R, server.R, or global.R:

```
shinyOptions(cache = diskCache(file.path(dirname(tempdir()), "myapp-cache")))
```

This will create a subdirectory in your system temp directory named myapp-cache (replace myapp-cache with a unique name of your choosing). On most platforms, this directory will be removed when your system reboots. This cache will persist across multiple starts and stops of the R process, as long as you do not reboot.

To have the cache persist even across multiple reboots, you can create the cache in a location outside of the temp directory. For example, it could be a subdirectory of the application:

```
shinyOptions(cache = diskCache("./myapp-cache"))
```

In this case, resetting the cache will have to be done manually, by deleting the directory.

You can also scope a cache to just one plot, or selected plots. To do that, create a memoryCache() or diskCache(), and pass it as the cache argument of renderCachedPlot.

### Interactive plots

renderCachedPlot can be used to create interactive plots. See plotOutput() for more information and examples.

### See Also

See renderPlot() for the regular, non-cached version of this function. For more about configuring caches, see memoryCache() and diskCache().

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

# A basic example that uses the default app-scoped memory cache.
# The cache will be shared among all simultaneous users of the application.
shinyApp(
  fluidPage(
    sidebarLayout(
      sidebarPanel(
        sliderInput("n", "Number of points", 4, 32, value = 8, step = 4)
      ),
      mainPanel(plotOutput("plot"))
    )
  ),
  function(input, output, session) {
    output$plot <- renderCachedPlot({
        Sys.sleep(2)  # Add an artificial delay
        seqn <- seq_len(input$n)
        plot(mtcars$wt[seqn], mtcars$mpg[seqn],
             xlim = range(mtcars$wt), ylim = range(mtcars$mpg))
      },
      cacheKeyExpr = { list(input$n) }
    )
  }
)


# An example uses a data object shared across sessions. mydata() is part of
# the cache key, so when its value changes, plots that were previously
# stored in the cache will no longer be used (unless mydata() changes back
# to its previous value).
mydata <- reactiveVal(data.frame(x = rnorm(400), y = rnorm(400)))

ui <- fluidPage(
  sidebarLayout(
```

```
      sidebarPanel(
        sliderInput("n", "Number of points", 50, 400, 100, step = 50),
        actionButton("newdata", "New data")
      ),
      mainPanel(
        plotOutput("plot")
      )
    )
  )

  server <- function(input, output, session) {
    observeEvent(input$newdata, {
      mydata(data.frame(x = rnorm(400), y = rnorm(400)))
    })

    output$plot <- renderCachedPlot(
      {
        Sys.sleep(2)
        d <- mydata()
        seqn <- seq_len(input$n)
        plot(d$x[seqn], d$y[seqn], xlim = range(d$x), ylim = range(d$y))
      },
      cacheKeyExpr = { list(input$n, mydata()) },
    )
  }

  shinyApp(ui, server)


  # A basic application with two plots, where each plot in each session has
  # a separate cache.
  shinyApp(
    fluidPage(
      sidebarLayout(
        sidebarPanel(
          sliderInput("n", "Number of points", 4, 32, value = 8, step = 4)
        ),
        mainPanel(
          plotOutput("plot1"),
          plotOutput("plot2")
        )
      )
    ),
    function(input, output, session) {
      output$plot1 <- renderCachedPlot({
          Sys.sleep(2)  # Add an artificial delay
          seqn <- seq_len(input$n)
          plot(mtcars$wt[seqn], mtcars$mpg[seqn],
               xlim = range(mtcars$wt), ylim = range(mtcars$mpg))
        },
        cacheKeyExpr = { list(input$n) },
        cache = memoryCache()
      )
```

```
        output$plot2 <- renderCachedPlot({
            Sys.sleep(2)  # Add an artificial delay
            seqn <- seq_len(input$n)
            plot(mtcars$wt[seqn], mtcars$mpg[seqn],
                 xlim = range(mtcars$wt), ylim = range(mtcars$mpg))
        },
        cacheKeyExpr = { list(input$n) },
        cache = memoryCache()
      )
    }
  )

}

## Not run:
# At the top of app.R, this set the application-scoped cache to be a memory
# cache that is 20 MB in size, and where cached objects expire after one
# hour.
shinyOptions(cache = memoryCache(max_size = 20e6, max_age = 3600))

# At the top of app.R, this set the application-scoped cache to be a disk
# cache that can be shared among multiple concurrent R processes, and is
# deleted when the system reboots.
shinyOptions(cache = diskCache(file.path(dirname(tempdir()), "myapp-cache"))

# At the top of app.R, this set the application-scoped cache to be a disk
# cache that can be shared among multiple concurrent R processes, and
# persists on disk across reboots.
shinyOptions(cache = diskCache("./myapp-cache"))

# At the top of the server function, this set the session-scoped cache to be
# a memory cache that is 5 MB in size.
server <- function(input, output, session) {
  shinyOptions(cache = memoryCache(max_size = 5e6))

  output$plot <- renderCachedPlot(
    ...,
    cache = "session"
  )
}


## End(Not run)
```

---

renderDataTable          *Table output with the JavaScript library DataTables*

---

### Description

Makes a reactive version of the given function that returns a data frame (or matrix), which will be
rendered with the DataTables library. Paging, searching, filtering, and sorting can be done on the R

side using Shiny as the server infrastructure.

## Usage

```
renderDataTable(expr, options = NULL, searchDelay = 500,
  callback = "function(oTable) {}", escape = TRUE,
  env = parent.frame(), quoted = FALSE, outputArgs = list())
```

## Arguments

| | |
|---|---|
| expr | An expression that returns a data frame or a matrix. |
| options | A list of initialization options to be passed to DataTables, or a function to return such a list. |
| searchDelay | The delay for searching, in milliseconds (to avoid too frequent search requests). |
| callback | A JavaScript function to be applied to the DataTable object. This is useful for DataTables plug-ins, which often require the DataTable instance to be available (<http://datatables.net/extensions/>). |
| escape | Whether to escape HTML entities in the table: TRUE means to escape the whole table, and FALSE means not to escape it. Alternatively, you can specify numeric column indices or column names to indicate which columns to escape, e.g. 1:5 (the first 5 columns), c(1,3,4), or c(-1,-3) (all columns except the first and third), or c('Species','Sepal.Length'). |
| 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. |
| outputArgs | A list of arguments to be passed through to the implicit call to [dataTableOutput()](#) when renderDataTable is used in an interactive R Markdown document. |

## Details

For the options argument, the character elements that have the class "AsIs" (usually returned from [base::I()](#)) will be evaluated in JavaScript. This is useful when the type of the option value is not supported in JSON, e.g., a JavaScript function, which can be obtained by evaluating a character string. Note this only applies to the root-level elements of the options list, and the I() notation does not work for lower-level elements in the list.

## Note

This function only provides the server-side version of DataTables (using R to process the data object on the server side). There is a separate package **DT** (<https://github.com/rstudio/DT>) that allows you to create both server-side and client-side DataTables, and supports additional DataTables features. Consider using DT::renderDataTable() and DT::dataTableOutput() (see [http://rstudio.github.io/DT/shiny.html](http://rstudio.github.io/DT/shiny.html) for more information).

## References

<http://datatables.net>

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # pass a callback function to DataTables using I()
  shinyApp(
    ui = fluidPage(
      fluidRow(
        column(12,
          dataTableOutput('table')
        )
      )
    ),
    server = function(input, output) {
      output$table <- renderDataTable(iris,
        options = list(
          pageLength = 5,
          initComplete = I("function(settings, json) {alert('Done.');}")
        )
      )
    }
  )
}
```

---

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, outputArgs = list())
```

### 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? Generally 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`. |
| outputArgs | A list of arguments to be passed through to the implicit call to `imageOutput()` when renderImage is used in an interactive R Markdown document. |

**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**

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

ui <- fluidPage(
  sliderInput("n", "Number of observations", 2, 1000, 500),
  plotOutput("plot1"),
  plotOutput("plot2"),
  plotOutput("plot3")
)

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

  # 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$plot2 <- renderImage({
    # Read plot2's width and height. These are reactive values, so this
    # expression will re-run whenever these values change.
    width  <- session$clientData$output_plot2_width
    height <- session$clientData$output_plot2_height
```

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

    png(outfile, width=width, height=height)
    hist(rnorm(input$n))
    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
  # NOTE: For this example to work, it would require files in a subdirectory
  # named images/
  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)
}

shinyApp(ui, server)
}
```

---

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, execOnResize = FALSE,
  outputArgs = list())
```

### Arguments

| | |
|---|---|
| expr | An expression that generates a plot. |
| width, height | Height and width can be specified in three ways: |

- "auto", the default, uses the size specified by `plotOutput()` (i.e. the `offsetWidth`/'offsetHeight" of the HTML element bound to this plot.)
- An integer, defining the width/height in pixels.
- A function that returns the width/height in pixels (or "auto"). The function is executed in a reactive context so that you can refer to reactive values and expression to make the width/height reactive.

  When rendering an inline plot, you must provide numeric values (in pixels) to both width and height.

res          Resolution of resulting plot, in pixels per inch. This value is passed to `grDevices::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 `grDevices::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.

execOnResize If FALSE (the default), then when a plot is resized, Shiny will *replay* the plot drawing commands with `grDevices::replayPlot()` instead of re-executing expr. This can result in faster plot redrawing, but there may be rare cases where it is undesirable. If you encounter problems when resizing a plot, you can have Shiny re-execute the code on resize by setting this to TRUE.

outputArgs   A list of arguments to be passed through to the implicit call to `plotOutput()` when renderPlot is used in an interactive R Markdown document.

### Details

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

### Interactive plots

With ggplot2 graphics, the code in renderPlot should return a ggplot object; if instead the code prints the ggplot2 object with something like print(p), then the coordinates for interactive graphics will not be properly scaled to the data space.

See `plotOutput()` for more information about interactive plots.

### See Also

For the corresponding client-side output function, and example usage, see `plotOutput()`. For more details on how the plots are generated, and how to control the output, see `plotPNG()`. `renderCachedPlot()` offers a way to cache generated plots to expedite the rendering of identical plots.

---

renderPrint                         *Printable Output*

---

### Description

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

### Usage

```
renderPrint(expr, env = parent.frame(), quoted = FALSE,
  width = getOption("width"), outputArgs = list())
```

### 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 is useful if you want to save an expression in a variable. |
| width | The value for `[options][base::options]('width')`. |
| outputArgs | A list of arguments to be passed through to the implicit call to `verbatimTextOutput()` when `renderPrint` is used in an interactive R Markdown document. |

### 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 `utils::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 `base::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, striped = FALSE, hover = FALSE, bordered = FALSE,
  spacing = c("s", "xs", "m", "l"), width = "auto", align = NULL,
  rownames = FALSE, colnames = TRUE, digits = NULL, na = "NA", ...,
  env = parent.frame(), quoted = FALSE, outputArgs = list())
```

### Arguments

| | |
|---|---|
| expr | An expression that returns an R object that can be used with [xtable::xtable()](). |
| striped, hover, bordered | |
| | Logicals: if TRUE, apply the corresponding Bootstrap table format to the output table. |
| spacing | The spacing between the rows of the table (xs stands for "extra small", s for "small", m for "medium" and l for "large"). |
| width | Table width. Must be a valid CSS unit (like "100 "auto") or a number, which will be coerced to a string and have "px" appended. |
| align | A string that specifies the column alignment. If equal to 'l', 'c' or 'r', then all columns will be, respectively, left-, center- or right-aligned. Otherwise, align must have the same number of characters as the resulting table (if rownames = TRUE, this will be equal to ncol()+1), with the *i*-th character specifying the alignment for the *i*-th column (besides 'l', 'c' and 'r', '?' is also permitted - '?' is a placeholder for that particular column, indicating that it should keep its default alignment). If NULL, then all numeric/integer columns (including the row names, if they are numbers) will be right-aligned and everything else will be left-aligned (align = '?' produces the same result). |
| rownames, colnames | |
| | Logicals: include rownames? include colnames (column headers)? |
| digits | An integer specifying the number of decimal places for the numeric columns (this will not apply to columns with an integer class). If digits is set to a negative value, then the numeric columns will be displayed in scientific format with a precision of abs(digits) digits. |
| na | The string to use in the table cells whose values are missing (i.e. they either evaluate to NA or NaN). |
| ... | Arguments to be passed through to [xtable::xtable()]() and [xtable::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. |
| outputArgs | A list of arguments to be passed through to the implicit call to [tableOutput()](#) when renderTable is used in an interactive R Markdown document. |

## 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 [base::cat()](#) to turn its result into a single-element character vector.

## Usage

```
renderText(expr, env = parent.frame(), quoted = FALSE,
  outputArgs = list(), sep = " ")
```

## 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. |
| outputArgs | A list of arguments to be passed through to the implicit call to [textOutput()](#) when renderText is used in an interactive R Markdown document. |
| sep | A separator passed to cat to be appended after each element. |

## 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 [utils::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

Renders reactive HTML using the Shiny UI library.

### Usage

```
renderUI(expr, env = parent.frame(), quoted = FALSE,
  outputArgs = list())
```

### 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. |
| outputArgs | A list of arguments to be passed through to the implicit call to [uiOutput()]() when renderUI is used in an interactive R Markdown document. |

### Details

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

### See Also

[uiOutput()]()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  uiOutput("moreControls")
)

server <- function(input, output) {
  output$moreControls <- renderUI({
    tagList(
      sliderInput("n", "N", 1, 1000, 500),
      textInput("label", "Label")
    )
  })
}
shinyApp(ui, server)
}
```

---

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 = stats::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 `base::.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
```

---

req                              *Check for required values*

---

## Description

Ensure that values are available ("truthy"–see Details) before proceeding with a calculation or action. If any of the given values is not truthy, the operation is stopped by raising a "silent" exception (not logged by Shiny, nor displayed in the Shiny app's UI).

## Usage

```
req(..., cancelOutput = FALSE)

isTruthy(x)
```

## Arguments

| | |
|---|---|
| ... | Values to check for truthiness. |
| cancelOutput | If TRUE and an output is being evaluated, stop processing as usual but instead of clearing the output, leave it in whatever state it happens to be in. |
| x | An expression whose truthiness value we want to determine |

## Details

The req function was designed to be used in one of two ways. The first is to call it like a statement (ignoring its return value) before attempting operations using the required values:

```
rv <- reactiveValues(state = FALSE)
r <- reactive({
  req(input$a, input$b, rv$state)
  # Code that uses input$a, input$b, and/or rv$state...
})
```

In this example, if r() is called and any of input$a, input$b, and rv$state are NULL, FALSE, "", etc., then the req call will trigger an error that propagates all the way up to whatever render block or observer is executing.

The second is to use it to wrap an expression that must be truthy:

```
output$plot <- renderPlot({
  if (req(input$plotType) == "histogram") {
    hist(dataset())
  } else if (input$plotType == "scatter") {
    qplot(dataset(), aes(x = x, y = y))
  }
})
```

In this example, `req(input$plotType)` first checks that `input$plotType` is truthy, and if so, returns it. This is a convenient way to check for a value "inline" with its first use.

**Truthy and falsy values**

The terms "truthy" and "falsy" generally indicate whether a value, when coerced to a `base::logical()`, is `TRUE` or `FALSE`. We use the term a little loosely here; our usage tries to match the intuitive notions of "Is this value missing or available?", or "Has the user provided an answer?", or in the case of action buttons, "Has the button been clicked?".

For example, a `textInput` that has not been filled out by the user has a value of `""`, so that is considered a falsy value.

To be precise, `req` considers a value truthy *unless* it is one of:

- `FALSE`
- `NULL`
- `""`
- An empty atomic vector
- An atomic vector that contains only missing values
- A logical vector that contains all `FALSE` or missing values
- An object of class `"try-error"`
- A value that represents an unclicked `actionButton()`

Note in particular that the value `0` is considered truthy, even though `as.logical(0)` is `FALSE`.

If the built-in rules for truthiness do not match your requirements, you can always work around them. Since `FALSE` is falsy, you can simply provide the results of your own checks to `req`:

`req(input$a != 0)`

**Using** `req(FALSE)`

You can use `req(FALSE)` (i.e. no condition) if you've already performed all the checks you needed to by that point and just want to stop the reactive chain now. There is no advantange to this, except perhaps ease of readability if you have a complicated condition to check for (or perhaps if you'd like to divide your condition into nested `if` statements).

**Using** `cancelOutput = TRUE`

When `req(...,cancelOutput = TRUE)` is used, the "silent" exception is also raised, but it is treated slightly differently if one or more outputs are currently being evaluated. In those cases, the reactive chain does not proceed or update, but the output(s) are left is whatever state they happen to be in (whatever was their last valid state).

Note that this is always going to be the case if this is used inside an output context (e.g. `output$txt <-...`). It may or may not be the case if it is used inside a non-output context (e.g. `reactive()`, `observe()` or `observeEvent()`) — depending on whether or not there is an `output$...` that is triggered as a result of those calls. See the examples below for concrete scenarios.

## Value

The first value that was passed in.

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    textInput('data', 'Enter a dataset from the "datasets" package', 'cars'),
    p('(E.g. "cars", "mtcars", "pressure", "faithful")'), hr(),
    tableOutput('tbl')
  )

  server <- function(input, output) {
    output$tbl <- renderTable({

      ## to require that the user types something, use: `req(input$data)`
      ## but better: require that input$data is valid and leave the last
      ## valid table up
      req(exists(input$data, "package:datasets", inherits = FALSE),
          cancelOutput = TRUE)

      head(get(input$data, "package:datasets", inherits = FALSE))
    })
  }

  shinyApp(ui, server)
}
```

---

| restoreInput | *Restore an input value* |
|---|---|

---

## Description

This restores an input value from the current restore context. It should be called early on inside of
input functions (like [textInput()](#)).

## Usage

```
restoreInput(id, default)
```

## Arguments

| id | Name of the input value to restore. |
|---|---|
| default | A default value to use, if there's no value to restore. |

---

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 = getOption("shiny.port"),
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"), workerId = "",
  quiet = FALSE, display.mode = c("auto", "normal", "showcase"),
  test.mode = getOption("shiny.testmode", FALSE))
```

### Arguments

| | |
|---|---|
| appDir | The application to run. Should be one of the following: |

- A directory containing server.R, plus, either ui.R or a www directory that contains the file index.html.
- A directory containing app.R.
- An .R file containing a Shiny application, ending with an expression that produces a Shiny app object.
- A list with ui and server components.
- A Shiny app object created by [shinyApp()](#).

| | |
|---|---|
| port | The TCP port that the application should listen on. If the port is not specified, and the shiny.port option is set (with options(shiny.port = XX)), then that port will be used. Otherwise, use a random port. |
| 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. This value of this parameter can also be a function to call with the application's URL. |
| host | The IPv4 address that the application should listen on. Defaults to the shiny.host option, if set, or "127.0.0.1" if not. See Details. |
| workerId | Can generally be ignored. Exists to help some editions of Shiny Server Pro route requests to the correct process. |
| quiet | Should Shiny status messages be shown? Defaults to FALSE. |
| display.mode | The mode in which to display the application. If set to the value "showcase", shows application code and metadata from a DESCRIPTION file in the application directory alongside the application. If set to "normal", displays the application normally. Defaults to "auto", which displays the application in the mode given in its DESCRIPTION file, if any. |
| test.mode | Should the application be launched in test mode? This is only used for recording or running automated tests. Defaults to the shiny.testmode option, or FALSE if the option is not set. |

**Details**

The host parameter was introduced in Shiny 0.9.0. Its default value of ″127.0.0.1″ means that, contrary to previous versions of Shiny, only the current machine can access locally hosted Shiny apps. To allow other clients to connect, use the value ″0.0.0.0″ instead (which was the value that was hard-coded into Shiny in 0.8.0 and earlier).

**Examples**

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

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

## End(Not run)

## Only run this example in interactive R sessions
if (interactive()) {
  options(device.ask.default = FALSE)

  # 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)) })
    }
  ))


  # Running a Shiny app object
  app <- shinyApp(
    ui = bootstrapPage(
      numericInput('n', 'Number of obs', 100),
      plotOutput('plot')
    ),
    server = function(input, output) {
      output$plot <- renderPlot({ hist(runif(input$n)) })
    }
  )
  runApp(app)
}
```

---

runExample                          *Run Shiny Example Applications*

---

### Description

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

### Usage

```
runExample(example = NA, port = NULL,
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"), display.mode = c("auto",
  "normal", "showcase"))
```

### 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 choosing a random port. |
| 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. |
| host | The IPv4 address that the application should listen on. Defaults to the shiny.host option, if set, or "127.0.0.1" if not. |
| display.mode | The mode in which to display the example. Defaults to showcase, but may be set to normal to see the example without code or commentary. |

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # 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")
}
```

---

runGadget                          *Run a gadget*

---

### Description

Similar to runApp, but handles input$cancel automatically, and if running in RStudio, defaults to viewing the app in the Viewer pane.

### Usage

```
runGadget(app, server = NULL, port = getOption("shiny.port"),
  viewer = paneViewer(), stopOnCancel = TRUE)
```

## Arguments

| | |
|---|---|
| app | Either a Shiny app object as created by [shinyApp()](shinyApp()) et al, or, a UI object. |
| server | Ignored if app is a Shiny app object; otherwise, passed along to shinyApp (i.e. shinyApp(ui = app, server = server)). |
| port | See [runApp()](runApp()). |
| viewer | Specify where the gadget should be displayed–viewer pane, dialog window, or external browser–by passing in a call to one of the [viewer()](viewer()) functions. |
| stopOnCancel | If TRUE (the default), then an observeEvent is automatically created that handles input$cancel by calling stopApp() with an error. Pass FALSE if you want to handle input$cancel yourself. |

## Value

The value returned by the gadget.

## Examples

```
## Not run:
library(shiny)

ui <- fillPage(...)

server <- function(input, output, session) {
  ...
}

# Either pass ui/server as separate arguments...
runGadget(ui, server)

# ...or as a single app object
runGadget(shinyApp(ui, server))

## End(Not run)
```

---

runUrl                          *Run a Shiny application from a URL*

---

## Description

runUrl() downloads and launches 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 the root directory or a subdirectory in the archive. For example, the files might be myapp/server.r and myapp/ui.r. The functions runGitHub() and runGist() are based on runUrl(), using URL's from GitHub (https://github.com) and GitHub gists (https://gist.github.com), respectively.

## Usage

```
runUrl(url, filetype = NULL, subdir = NULL, destdir = NULL, ...)

runGist(gist, destdir = NULL, ...)

runGitHub(repo, username = getOption("github.user"), ref = "master",
  subdir = NULL, destdir = NULL, ...)
```

## 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". |
| destdir | Directory to store the downloaded application files. If NULL (the default), the application files will be stored in a temporary directory and removed when the app exits |
| ... | Other arguments to be passed to [runApp()](#), such as port and launch.browser. |
| 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. |
| repo | Name of the repository. |
| username | GitHub username. If repo is of the form "username/repo", username will be taken from repo. |
| ref | Desired git reference. Could be a commit, tag, or branch name. Defaults to "master". |

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  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/")
}
## Only run this example in interactive R sessions
if (interactive()) {
  runGist(3239667)
  runGist("https://gist.github.com/jcheng5/3239667")

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

```
## Only run this example in interactive R sessions
if (interactive()) {
  runGitHub("shiny_example", "rstudio")
  # or runGitHub("rstudio/shiny_example")

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

---

safeError                          *Declare an error safe for the user to see*

---

### Description

This should be used when you want to let the user see an error message even if the default is to
sanitize all errors. If you have an error e and call stop(safeError(e)), then Shiny will ignore the
value of getOption("shiny.sanitize.errors") and always display the error in the app itself.

### Usage

```
safeError(error)
```

### Arguments

error          Either an "error" object or a "character" object (string). In the latter case, the
               string will become the message of the error returned by safeError.

### Details

An error generated by safeError has priority over all other Shiny errors. This can be dangerous.
For example, if you have set options(shiny.sanitize.errors = TRUE), then by default all error
messages are omitted in the app, and replaced by a generic error message. However, this does not
apply to safeError: whatever you pass through error will be displayed to the user. So, this should
only be used when you are sure that your error message does not contain any sensitive information.
In those situations, safeError can make your users' lives much easier by giving them a hint as to
where the error occurred.

### Value

An "error" object

### See Also

[shiny-options()](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

# uncomment the desired line to experiment with shiny.sanitize.errors
# options(shiny.sanitize.errors = TRUE)
# options(shiny.sanitize.errors = FALSE)

# Define UI
ui <- fluidPage(
  textInput('number', 'Enter your favorite number from 1 to 10', '5'),
  textOutput('normalError'),
  textOutput('safeError')
)

# Server logic
server <- function(input, output) {
  output$normalError <- renderText({
    number <- input$number
    if (number %in% 1:10) {
      return(paste('You chose', number, '!'))
    } else {
      stop(
        paste(number, 'is not a number between 1 and 10')
      )
    }
  })
  output$safeError <- renderText({
    number <- input$number
    if (number %in% 1:10) {
      return(paste('You chose', number, '!'))
    } else {
      stop(safeError(
        paste(number, 'is not a number between 1 and 10')
      ))
    }
  })
}

# Complete app with UI and server components
shinyApp(ui, server)
}
```

---

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,
  selectize = TRUE, width = NULL, size = NULL)

selectizeInput(inputId, ..., options = NULL, width = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the group labels (leveraging the <optgroup> HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature. |
| selected | The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists. |
| multiple | Is selection of multiple items allowed? |
| selectize | Whether to use **selectize.js** or not. |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](). |
| size | Number of items to show in the selection box; a larger number will result in a taller box. Not compatible with selectize=TRUE. Normally, when multiple=FALSE, a select input will be a drop-down list, but when size is set, it will be a box instead. |
| ... | Arguments passed to selectInput(). |
| options | A list of options. See the documentation of **selectize.js** for possible options (character option values inside [base::I()]() will be treated as literal JavaScript code; see [renderDataTable()]() for details). |

## Details

By default, selectInput() and selectizeInput() use the JavaScript library **selectize.js** ([https://github.com/selectize/selectize.js](https://github.com/selectize/selectize.js)) instead of the basic select input element. To use the standard HTML select input element, use selectInput() with selectize=FALSE.

In selectize mode, if the first element in choices has a value of "", its name will be treated as a placeholder prompt. For example: selectInput("letter","Letter",c("Choose one" = "",LETTERS))

## Value

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

**Server value**

A vector of character strings, usually of length 1, with the value of the selected items. When `multiple=TRUE` and nothing is selected, this value will be `NULL`.

**Note**

The selectize input created from `selectizeInput()` allows deletion of the selected option even in a single select input, which will return an empty string as its value. This is the default behavior of **selectize.js**. However, the selectize input created from `selectInput(...,selectize = TRUE)` will ignore the empty string value when it is a single choice input and the empty string is not in the `choices` argument. This is to keep compatibility with `selectInput(...,selectize = FALSE)`.

**See Also**

`updateSelectInput()` `varSelectInput()`

Other input elements: `actionButton`, `checkboxGroupInput`, `checkboxInput`, `dateInput`, `dateRangeInput`, `fileInput`, `numericInput`, `passwordInput`, `radioButtons`, `sliderInput`, `submitButton`, `textAreaInput`, `textInput`, `varSelectInput`

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

# basic example
shinyApp(
  ui = fluidPage(
    selectInput("variable", "Variable:",
                c("Cylinders" = "cyl",
                  "Transmission" = "am",
                  "Gears" = "gear")),
    tableOutput("data")
  ),
  server = function(input, output) {
    output$data <- renderTable({
      mtcars[, c("mpg", input$variable), drop = FALSE]
    }, rownames = TRUE)
  }
)

# demoing group support in the `choices` arg
shinyApp(
  ui = fluidPage(
    selectInput("state", "Choose a state:",
      list(`East Coast` = list("NY", "NJ", "CT"),
           `West Coast` = list("WA", "OR", "CA"),
           `Midwest` = list("MN", "WI", "IA"))
    ),
    textOutput("result")
  ),
  server = function(input, output) {
```

```
    output$result <- renderText({
      paste("You chose", input$state)
    })
  }
)
}
```

---

serverInfo                    *Collect information about the Shiny Server environment*

---

### Description

This function returns the information about the current Shiny Server, such as its version, and whether it is the open source edition or professional edition. If the app is not served through the Shiny Server, this function just returns `list(shinyServer = FALSE)`.

### Usage

```
serverInfo()
```

### Details

This function will only return meaningful data when using Shiny Server version 1.2.2 or later.

### Value

A list of the Shiny Server information.

---

session                    *Session object*

---

### Description

Shiny server functions can optionally include `session` as a parameter (e.g. `function(input,output,session)`). The session object is an environment that can be used to access information and functionality relating to the session. The following list describes the items available in the environment; they can be accessed using the $ operator (for example, `session$clientData$url_search`).

**Value**

allowReconnect(value)

> If value is TRUE and run in a hosting environment (Shiny Server or Connect) with reconnections enabled, then when the session ends due to the network connection closing, the client will attempt to reconnect to the server. If a reconnection is successful, the browser will send all the current input values to the new session on the server, and the server will recalculate any outputs and send them back to the client. If value is FALSE, reconnections will be disabled (this is the default state). If "force", then the client browser will always attempt to reconnect. The only reason to use "force" is for testing on a local connection (without Shiny Server or Connect).

clientData   A [reactiveValues()](#) object that contains information about the client.

> - allowDataUriScheme is a logical value that indicates whether the browser is able to handle URIs that use the data: scheme.
> - pixelratio reports the "device pixel ratio" from the web browser, or 1 if none is reported. The value is 2 for Apple Retina displays.
> - singletons - for internal use
> - url_protocol, url_hostname, url_port, url_pathname, url_search, url_hash_initial and url_hash can be used to get the components of the URL that was requested by the browser to load the Shiny app page. These values are from the browser's perspective, so neither HTTP proxies nor Shiny Server will affect these values. The url_search value may be used with [parseQueryString()](#) to access query string parameters.

> clientData also contains information about each output. output_*outputId*_width and output_*outputId*_height give the dimensions (using offsetWidth and offsetHeight) of the DOM element that is bound to *outputId*, and output_*outputId*_hidden is a logical that indicates whether the element is hidden. These values may be NULL if the output is not bound.

input        The session's input object (the same as is passed into the Shiny server function as an argument).

isClosed()   A function that returns TRUE if the client has disconnected.

ns(id)       Server-side version of [ns <-NS(id)](#). If bare IDs need to be explicitly namespaced for the current module, session$ns("name") will return the fully-qualified ID.

onEnded(callback)

> Synonym for onSessionEnded.

onFlush(func, once=TRUE)

> Registers a function to be called before the next time (if once=TRUE) or every time (if once=FALSE) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

onFlushed(func, once=TRUE)

> Registers a function to be called after the next time (if once=TRUE) or every time (if once=FALSE) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

onSessionEnded(callback)

> Registers a function to be called after the client has disconnected. Returns a
> function that can be called with no arguments to cancel the registration.

output           The session's output object (the same as is passed into the Shiny server function
                 as an argument).

reactlog         For internal use.

registerDataObj(name, data, filterFunc)

> Publishes any R object as a URL endpoint that is unique to this session. name
> must be a single element character vector; it will be used to form part of the
> URL. filterFunc must be a function that takes two arguments: data (the value
> that was passed into registerDataObj) and req (an environment that imple-
> ments the Rook specification for HTTP requests). filterFunc will be called
> with these values whenever an HTTP request is made to the URL endpoint. The
> return value of filterFunc should be a Rook-style response.

reload()         The equivalent of hitting the browser's Reload button. Only works if the session
                 is actually connected.

request          An environment that implements the Rook specification for HTTP requests.
                 This is the request that was used to initiate the websocket connection (as op-
                 posed to the request that downloaded the web page for the app).

userData         An environment for app authors and module/package authors to store whatever
                 session-specific data they want.

resetBrush(brushId)

> Resets/clears the brush with the given brushId, if it exists on any imageOutput
> or plotOutput in the app.

sendCustomMessage(type, message)

> Sends a custom message to the web page. type must be a single-element char-
> acter vector giving the type of message, while message can be any jsonlite-
> encodable value. Custom messages have no meaning to Shiny itself; they are
> used soley to convey information to custom JavaScript logic in the browser. You
> can do this by adding JavaScript code to the browser that calls Shiny.addCustomMessageHandler(type,
> as the page loads; the function you provide to addCustomMessageHandler will
> be invoked each time sendCustomMessage is called on the server.

sendBinaryMessage(type, message)

> Similar to sendCustomMessage, but the message must be a raw vector and the
> registration method on the client is Shiny.addBinaryMessageHandler(type, function(message){...
> The message argument on the client will be a [DataView].

sendInputMessage(inputId, message)

> Sends a message to an input on the session's client web page; if the input is
> present and bound on the page at the time the message is received, then the
> input binding object's receiveMessage(el, message) method will be called.
> sendInputMessage should generally not be called directly from Shiny apps,
> but through friendlier wrapper functions like [updateTextInput()].

setBookmarkExclude(names)

> Set input names to be excluded from bookmarking.

getBookmarkExclude()

> Returns the set of input names to be excluded from bookmarking.

onBookmark(fun)

> Registers a function that will be called just before bookmarking state.

onBookmarked(fun)

> Registers a function that will be called just after bookmarking state.

onRestore(fun)   Registers a function that will be called when a session is restored, before all other reactives, observers, and render functions are run.

onRestored(fun)

> Registers a function that will be called when a session is restored, after all other reactives, observers, and render functions are run.

doBookmark()   Do bookmarking and invoke the onBookmark and onBookmarked callback functions.

exportTestValues()

> Registers expressions for export in test mode, available at the test snapshot URL.

getTestSnapshotUrl(input=TRUE, output=TRUE, export=TRUE, format="json")

> Returns a URL for the test snapshots. Only has an effect when the `shiny.testmode` option is set to TRUE. For the input, output, and export arguments, TRUE means to return all of these values. It is also possible to specify by name which values to return by providing a character vector, as in `input=c("x","y")`. The format can be "rds" or "json".

---

setBookmarkExclude   *Exclude inputs from bookmarking*

---

## Description

This function tells Shiny which inputs should be excluded from bookmarking. It should be called from inside the application's server function.

## Usage

```
setBookmarkExclude(names = character(0),
  session = getDefaultReactiveDomain())
```

## Arguments

names        A character vector containing names of inputs to exclude from bookmarking.

session      A shiny session object.

## Details

This function can also be called from a module's server function, in which case it will exclude inputs with the specified names, from that module. It will not affect inputs from other modules or from the top level of the Shiny application.

## See Also

[enableBookmarking()](#) for examples.

## shinyApp                    *Create a Shiny app object*

### Description

These functions create Shiny app objects from either an explicit UI/server pair (shinyApp), or by
passing the path of a directory that contains a Shiny app (shinyAppDir).

### Usage

```
shinyApp(ui, server, onStart = NULL, options = list(),
  uiPattern = "/", enableBookmarking = NULL)

shinyAppDir(appDir, options = list())

shinyAppFile(appFile, options = list())
```

### Arguments

ui                The UI definition of the app (for example, a call to fluidPage() with nested
                  controls)

server            A function with three parameters: input, output, and session. The function
                  is called once for each session ensuring that each app is independent.

onStart           A function that will be called before the app is actually run. This is only needed
                  for shinyAppObj, since in the shinyAppDir case, a global.R file can be used
                  for this purpose.

options           Named options that should be passed to the runApp call (these can be any of
                  the following: "port", "launch.browser", "host", "quiet", "display.mode" and
                  "test.mode"). You can also specify width and height parameters which pro-
                  vide a hint to the embedding environment about the ideal height/width for the
                  app.

uiPattern         A regular expression that will be applied to each GET request to determine whether
                  the ui should be used to handle the request. Note that the entire request path
                  must match the regular expression in order for the match to be considered suc-
                  cessful.

enableBookmarking
                  Can be one of "url", "server", or "disable". This is equivalent to calling
                  the enableBookmarking() function just before calling shinyApp(). With the
                  default value (NULL), the app will respect the setting from any previous calls to
                  enableBookmarking(). See enableBookmarking() for more information.

appDir            Path to directory that contains a Shiny app (i.e. a server.R file and either ui.R or
                  www/index.html)

appFile           Path to a .R file containing a Shiny application

## Details

Normally when this function is used at the R console, the Shiny app object is automatically passed to the `print()` function, which runs the app. If this is called in the middle of a function, the value will not be passed to `print()` and the app will not be run. To make the app run, pass the app object to `print()` or `runApp()`.

## Value

An object that represents the app. Printing the object or passing it to `runApp()` will run the app.

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  options(device.ask.default = FALSE)

  shinyApp(
    ui = fluidPage(
      numericInput("n", "n", 1),
      plotOutput("plot")
    ),
    server = function(input, output) {
      output$plot <- renderPlot( plot(head(cars, input$n)) )
    }
  )

  shinyAppDir(system.file("examples/01_hello", package="shiny"))


  # The object can be passed to runApp()
  app <- shinyApp(
    ui = fluidPage(
      numericInput("n", "n", 1),
      plotOutput("plot")
    ),
    server = function(input, output) {
      output$plot <- renderPlot( plot(head(cars, input$n)) )
    }
  )

  runApp(app)
}
```

---

showBookmarkUrlModal     *Display a modal dialog for bookmarking*

---

**Description**

This is a wrapper function for [urlModal()](#) that is automatically called if an application is book-marked but no other [onBookmark()](#) callback was set. It displays a modal dialog with the bookmark URL, along with a subtitle that is appropriate for the type of bookmarking used ("url" or "server").

**Usage**

```
showBookmarkUrlModal(url)
```

**Arguments**

url     A URL to show in the modal dialog.

---

showModal       *Show or remove a modal dialog*

---

**Description**

This causes a modal dialog to be displayed in the client browser, and is typically used with [modalDialog()](#).

**Usage**

```
showModal(ui, session = getDefaultReactiveDomain())

removeModal(session = getDefaultReactiveDomain())
```

**Arguments**

ui     UI content to show in the modal.

session    The session object passed to function given to shinyServer.

**See Also**

[modalDialog()](#) for examples.

---

showNotification                  *Show or remove a notification*

---

### Description

These functions show and remove notifications in a Shiny application.

### Usage

```
showNotification(ui, action = NULL, duration = 5, closeButton = TRUE,
  id = NULL, type = c("default", "message", "warning", "error"),
  session = getDefaultReactiveDomain())

removeNotification(id, session = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| ui | Content of message. |
| action | Message content that represents an action. For example, this could be a link that the user can click on. This is separate from ui so customized layouts can handle the main notification content separately from action content. |
| duration | Number of seconds to display the message before it disappears. Use NULL to make the message not automatically disappear. |
| closeButton | If TRUE, display a button which will make the notification disappear when clicked. If FALSE do not display. |
| id | A unique identifier for the notification. |
| | id is optional for showNotification(): Shiny will automatically create one if needed. If you do supply it, Shiny will update an existing notification if it exists, otherwise it will create a new one. |
| | id is required for removeNotification(). |
| type | A string which controls the color of the notification. One of "default" (gray), "message" (blue), "warning" (yellow), or "error" (red). |
| session | Session object to send notification to. |

### Value

An ID for the notification.

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
# Show a message when button is clicked
shinyApp(
  ui = fluidPage(
    actionButton("show", "Show")
```

```
    ),
    server = function(input, output) {
      observeEvent(input$show, {
        showNotification("Message text",
          action = a(href = "javascript:location.reload();", "Reload page")
        )
      })
    }
  )

  # App with show and remove buttons
  shinyApp(
    ui = fluidPage(
      actionButton("show", "Show"),
      actionButton("remove", "Remove")
    ),
    server = function(input, output) {
      # A queue of notification IDs
      ids <- character(0)
      # A counter
      n <- 0

      observeEvent(input$show, {
        # Save the ID for removal later
        id <- showNotification(paste("Message", n), duration = NULL)
        ids <<- c(ids, id)
        n <<- n + 1
      })

      observeEvent(input$remove, {
        if (length(ids) > 0)
          removeNotification(ids[1])
        ids <<- ids[-1]
      })
    }
  )
  }
```

---

showTab                         *Dynamically hide/show a tabPanel*

---

### Description

Dynamically hide or show a [tabPanel()](#) (or a [navbarMenu()](#))from an existing [tabsetPanel()](#), [navlistPanel()](#) or [navbarPage()](#).

### Usage

```
showTab(inputId, target, select = FALSE,
  session = getDefaultReactiveDomain())
```

```
hideTab(inputId, target, session = getDefaultReactiveDomain())
```

## Arguments

| inputId | The id of the `tabsetPanel` (or `navlistPanel` or `navbarPage`) in which to find `target`. |
|---------|---------------------------------------------------------------------------------------------|
| target  | The value of the `tabPanel` to be hidden/shown. See Details if you want to hide/show an entire `navbarMenu` instead. |
| select  | Should `target` be selected upon being shown? |
| session | The shiny session within which to call this function. |

## Details

For `navbarPage`, you can hide/show conventional `tabPanels` (whether at the top level or nested inside a `navbarMenu`), as well as an entire [navbarMenu()](#). For the latter case, `target` should be the menuName that you gave your `navbarMenu` when you first created it (by default, this is equal to the value of the `title` argument).

## See Also

[insertTab()](#)

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {

ui <- navbarPage("Navbar page", id = "tabs",
  tabPanel("Home",
    actionButton("hideTab", "Hide 'Foo' tab"),
    actionButton("showTab", "Show 'Foo' tab"),
    actionButton("hideMenu", "Hide 'More' navbarMenu"),
    actionButton("showMenu", "Show 'More' navbarMenu")
  ),
  tabPanel("Foo", "This is the foo tab"),
  tabPanel("Bar", "This is the bar tab"),
  navbarMenu("More",
    tabPanel("Table", "Table page"),
    tabPanel("About", "About page"),
    "------",
    "Even more!",
    tabPanel("Email", "Email page")
  )
)

server <- function(input, output, session) {
  observeEvent(input$hideTab, {
    hideTab(inputId = "tabs", target = "Foo")
  })
```

```
  observeEvent(input$showTab, {
    showTab(inputId = "tabs", target = "Foo")
  })

  observeEvent(input$hideMenu, {
    hideTab(inputId = "tabs", target = "More")
  })

  observeEvent(input$showMenu, {
    showTab(inputId = "tabs", target = "More")
  })
}

shinyApp(ui, server)
}
```

---

sidebarLayout                     *Layout a sidebar and main area*

---

### Description

Create a layout (sidebarLayout()) with a sidebar (sidebarPanel()) and main area (mainPanel()).
The sidebar is displayed with a distinct background color and typically contains input controls. The
main area occupies 2/3 of the horizontal width and typically contains outputs.

### Usage

```
sidebarLayout(sidebarPanel, mainPanel, position = c("left", "right"),
  fluid = TRUE)

sidebarPanel(..., width = 4)

mainPanel(..., width = 8)
```

### Arguments

| | |
|---|---|
| sidebarPanel | The sidebarPanel() containing input controls. |
| mainPanel | The mainPanel() containing outputs. |
| position | The position of the sidebar relative to the main area ("left" or "right"). |
| fluid | TRUE to use fluid layout; FALSE to use fixed layout. |
| ... | Output elements to include in the sidebar/main panel. |
| width | The width of the sidebar and main panel. By default, the sidebar takes up 1/3 of the width, and the main panel 2/3. The total width must be 12 or less. |

## See Also

Other layout functions: `fillPage`, `fixedPage`, `flowLayout`, `fluidPage`, `navbarPage`, `splitLayout`, `verticalLayout`

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

# Define UI
ui <- fluidPage(

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

  sidebarLayout(

    # 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")
    )
  )
)

# Server logic
server <- function(input, output) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}

# Complete app with UI and server components
shinyApp(ui, server)
}
```

| 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.

## Usage

```
singleton(x, value = TRUE)

is.singleton(x)
```

## Arguments

| | |
|---|---|
| x | A [tag](), text, [HTML](), or list. |
| value | Whether the object should be a singleton. |

---

| sizeGrowthRatio | *Create a sizing function that grows at a given ratio* |
|---|---|

---

## Description

Returns a function which takes a two-element vector representing an input width and height, and
returns a two-element vector of width and height. The possible widths are the base width times the
growthRate to any integer power. For example, with a base width of 500 and growth rate of 1.25,
the possible widths include 320, 400, 500, 625, 782, and so on, both smaller and larger. Sizes are
rounded up to the next pixel. Heights are computed the same way as widths.

## Usage

```
sizeGrowthRatio(width = 400, height = 400, growthRate = 1.2)
```

## Arguments

| | |
|---|---|
| width, height | Base width and height. |
| growthRate | Growth rate multiplier. |

## See Also

This is to be used with [renderCachedPlot()]().

## Examples

```
f <- sizeGrowthRatio(500, 500, 1.25)
f(c(400, 400))
f(c(500, 500))
f(c(530, 550))
f(c(625, 700))
```

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 = NULL, locale = NULL, ticks = TRUE,
  animate = FALSE, width = NULL, sep = ",", pre = NULL,
  post = NULL, timeFormat = NULL, timezone = NULL,
  dragRange = TRUE)

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

### Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| 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 (if NULL, a heuristic is used to determine the step size). If the values are dates, step is in days; if the values are times (POSIXt), step is in seconds. |
| 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 | Deprecated. |
| locale | Deprecated. |
| 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()](#). |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](#). |
| sep | Separator between thousands places in numbers. |
| pre | A prefix string to put in front of the value. |

| | |
|---|---|
| post | A suffix string to put after the value. |
| timeFormat | Only used if the values are Date or POSIXt objects. A time format string, to be passed to the Javascript strftime library. See https://github.com/samsonjs/strftime for more details. The allowed format specifications are very similar, but not identical, to those for R's base::strftime() function. For Dates, the default is "%F" (like "2015-07-01"), and for POSIXt, the default is "%F %T" (like "2015-07-01 15:32:10"). |
| timezone | Only used if the values are POSIXt objects. A string specifying the time zone offset for the displayed times, in the format "+HHMM" or "-HHMM". If NULL (the default), times will be displayed in the browser's time zone. The value "+0000" will result in UTC time. |
| dragRange | This option is used only if it is a range slider (with two values). If TRUE (the default), the range can be dragged. In other words, the min and max can be dragged together. If FALSE, the range cannot be dragged. |
| 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. |

## Server value

A number, or in the case of slider range, a vector of two numbers.

## See Also

updateSliderInput()

Other input elements: actionButton, checkboxGroupInput, checkboxInput, dateInput, dateRangeInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, submitButton, textAreaInput, textInput, varSelectInput

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

ui <- fluidPage(
  sliderInput("obs", "Number of observations:",
    min = 0, max = 1000, value = 500
  ),
  plotOutput("distPlot")
)

# Server logic
server <- function(input, output) {
  output$distPlot <- renderPlot({
```

```
    hist(rnorm(input$obs))
  })
}

# Complete app with UI and server components
shinyApp(ui, server)
}
```

---

snapshotExclude *Mark an output to be excluded from test snapshots*

---

### Description

Mark an output to be excluded from test snapshots

### Usage

```
snapshotExclude(x)
```

### Arguments

x               A reactive which will be assigned to an output.

---

snapshotPreprocessInput

*Add a function for preprocessing an input before taking a test snapshot*

---

### Description

Add a function for preprocessing an input before taking a test snapshot

### Usage

```
snapshotPreprocessInput(inputId, fun,
  session = getDefaultReactiveDomain())
```

### Arguments

inputId         Name of the input value.

fun             A function that takes the input value and returns a modified value. The returned
                value will be used for the test snapshot.

session         A Shiny session object.

snapshotPreprocessOutput

*Add a function for preprocessing an output before taking a test snapshot*

### Description

Add a function for preprocessing an output before taking a test snapshot

### Usage

```
snapshotPreprocessOutput(x, fun)
```

### Arguments

x           A reactive which will be assigned to an output.

fun         A function that takes the output value as an input and returns a modified value.
            The returned value will be used for the test snapshot.

splitLayout                 *Split layout*

### Description

Lays out elements horizontally, dividing the available horizontal space into equal parts (by default).

### Usage

```
splitLayout(..., cellWidths = NULL, cellArgs = list())
```

### Arguments

...         Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag.

cellWidths  Character or numeric vector indicating the widths of the individual cells. Recycling will be used if needed. Character values will be interpreted as CSS lengths (see [validateCssUnit()](#)), numeric values as pixels.

cellArgs    Any additional attributes that should be used for each cell of the layout.

### See Also

Other layout functions: [fillPage](#), [fixedPage](#), [flowLayout](#), [fluidPage](#), [navbarPage](#), [sidebarLayout](#), [verticalLayout](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

# Server code used for all examples
server <- function(input, output) {
  output$plot1 <- renderPlot(plot(cars))
  output$plot2 <- renderPlot(plot(pressure))
  output$plot3 <- renderPlot(plot(AirPassengers))
}

# Equal sizing
ui <- splitLayout(
  plotOutput("plot1"),
  plotOutput("plot2")
)
shinyApp(ui, server)

# Custom widths
ui <- splitLayout(cellWidths = c("25%", "75%"),
  plotOutput("plot1"),
  plotOutput("plot2")
)
shinyApp(ui, server)

# All cells at 300 pixels wide, with cell padding
# and a border around everything
ui <- splitLayout(
  style = "border: 1px solid silver;",
  cellWidths = 300,
  cellArgs = list(style = "padding: 6px"),
  plotOutput("plot1"),
  plotOutput("plot2"),
  plotOutput("plot3")
)
shinyApp(ui, server)
}
```

---

stopApp | *Stop the currently running Shiny app*

---

### Description

Stops the currently running Shiny app, returning control to the caller of [runApp()](#).

### Usage

```
stopApp(returnValue = invisible())
```

**Arguments**

returnValue        The value that should be returned from [runApp()](#).

---

submitButton                          *Create a submit button*

---

**Description**

Create a submit button for an app. Apps 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. The use of submitButton is generally discouraged in favor of the more versatile [actionButton()](#) (see details below).

**Usage**

```
submitButton(text = "Apply Changes", icon = NULL, width = NULL)
```

**Arguments**

text               Button caption

icon               Optional [icon()](#) to appear on the button

width              The width of the button, e.g. '400px', or '100%'; see [validateCssUnit()](#).

**Details**

Submit buttons are unusual Shiny inputs, and we recommend using [actionButton()](#) instead of submitButton when you want to delay a reaction. See [this article](#) for more information (including a demo of how to "translate" code using a submitButton to code using an actionButton).

In essence, the presence of a submit button stops all inputs from sending their values automatically to the server. This means, for instance, that if there are *two* submit buttons in the same app, clicking either one will cause all inputs in the app to send their values to the server. This is probably not what you'd want, which is why submit button are unwieldy for all but the simplest apps. There are other problems with submit buttons: for example, dynamically created submit buttons (for example, with [renderUI()](#) or [insertUI()](#)) will not work.

**Value**

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

**See Also**

Other input elements: [actionButton](#), [checkboxGroupInput](#), [checkboxInput](#), [dateInput](#), [dateRangeInput](#), [fileInput](#), [numericInput](#), [passwordInput](#), [radioButtons](#), [selectInput](#), [sliderInput](#), [textAreaInput](#), [textInput](#), [varSelectInput](#)

## Examples

```
if (interactive()) {

shinyApp(
  ui = basicPage(
    numericInput("num", label = "Make changes", value = 1),
    submitButton("Update View", icon("refresh")),
    helpText("When you click the button above, you should see",
             "the output below update to reflect the value you",
             "entered at the top:"),
    verbatimTextOutput("value")
  ),
  server = function(input, output) {

    # submit buttons do not have a value of their own,
    # they control when the app accesses values of other widgets.
    # input$num is the value of the number widget.
    output$value <- renderPrint({ input$num })
  }
)
}
```

suppressDependencies    *Suppress web dependencies*

## Description

This suppresses one or more web dependencies. It is meant to be used when a dependency (like a JavaScript or CSS file) is declared in raw HTML, in an HTML template.

## Usage

```
suppressDependencies(...)
```

## Arguments

| | |
|---|---|
| ... | Names of the dependencies to suppress. For example, "jquery" or "bootstrap". |

## See Also

htmlTemplate for more information about using HTML templates.

htmlDependency

---

tableOutput          *Create a table output element*

---

### Description

Render a renderTable() or renderDataTable() within an application page. renderTable uses a
standard HTML table, while renderDataTable uses the DataTables Javascript library to create an
interactive table with more features.

### Usage

```
tableOutput(outputId)

dataTableOutput(outputId)
```

### Arguments

outputId       output variable to read the table from

### Value

A table output element that can be included in a panel

### See Also

renderTable(), renderDataTable().

### Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # table example
  shinyApp(
    ui = fluidPage(
      fluidRow(
        column(12,
          tableOutput('table')
        )
      )
    ),
    server = function(input, output) {
      output$table <- renderTable(iris)
    }
  )


  # DataTables example
  shinyApp(
    ui = fluidPage(
```

```
      fluidRow(
        column(12,
          dataTableOutput('table')
        )
      )
    ),
    server = function(input, output) {
      output$table <- renderDataTable(iris)
    }
  )
}
```

tabPanel                    *Create a tab panel*

### Description

Create a tab panel that can be included within a [tabsetPanel()](#) or a [navbarPage()](#).

### Usage

```
tabPanel(title, ..., value = title, icon = 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.. |
| icon | Optional icon to appear on the tab. This attribute is only valid when using a tabPanel within a [navbarPage()](#). |

### 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, type = c("tabs",
  "pills"), position = 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. |
| type | Use "tabs" for the standard look; Use "pills" for a more plain look where tabs are selected using a background fill color. |
| position | This argument is deprecated; it has been discontinued in Bootstrap 3. |

### Value

A tabset that can be passed to [mainPanel()](#)

### See Also

[tabPanel()](#), [updateTabsetPanel()](#), [insertTab()](#), [showTab()](#)

### 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 de-
fined 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.

## Usage

```
tagList(...)

tagAppendAttributes(tag, ...)

tagHasAttribute(tag, attr)

tagGetAttribute(tag, attr)

tagAppendChild(tag, child)

tagAppendChildren(tag, ..., list = NULL)

tagSetChildren(tag, ..., list = NULL)

tag(`_tag_name`, varArgs, .noWS = NULL)
```

## Arguments

| | |
|---|---|
| `...` | Unnamed items that comprise this list of tags. |
| `tag` | A tag to append child elements to. |
| `attr` | The name of an attribute. |
| `child` | A child element to append to a parent tag. |
| `list` | An optional list of elements. Can be used with or instead of the . . . items. |
| `_tag_name` | HTML tag name |
| `varArgs` | List of attributes and children of the element. Named list items become at-tributes, 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. |
| `.noWS` | Character vector used to omit some of the whitespace that would normally be written around this tag. Valid options include `before`, `after`, `outside`, `after-begin`, and `before-end`. Any number of these options can be specified. |

## 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)

# suppress the whitespace between tags
oneline <- tag("span",
  tag("strong", "Super strong", .noWS="outside")
)
cat(as.character(oneline))
```

---

textAreaInput                          *Create a textarea input control*

---

## Description

Create a textarea input control for entry of unstructured text values.

## Usage

```
textAreaInput(inputId, label, value = "", width = NULL,
  height = NULL, cols = NULL, rows = NULL, placeholder = NULL,
  resize = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The `input` slot that will be used to access the value. |
| label | Display label for the control, or `NULL` for no label. |
| value | Initial value. |
| width | The width of the input, e.g. `'400px'`, or `'100%'`; see `validateCssUnit()`. |
| height | The height of the input, e.g. `'400px'`, or `'100%'`; see `validateCssUnit()`. |
| cols | Value of the visible character columns of the input, e.g. 80. This argument will only take effect if there is not a CSS `width` rule defined for this element; such a rule could come from the `width` argument of this function or from a containing page layout such as `fluidPage()`. |

| | |
|---|---|
| rows | The value of the visible character rows of the input, e.g. 6. If the `height` argument is specified, `height` will take precedence in the browser's rendering. |
| placeholder | A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option. |
| resize | Which directions the textarea box can be resized. Can be one of `"both"`, `"none"`, `"vertical"`, and `"horizontal"`. The default, `NULL`, will use the client browser's default setting for resizing textareas. |

### Value

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

### Server value

A character string of the text input. The default value is `""` unless `value` is provided.

### See Also

[updateTextAreaInput()](updateTextAreaInput())

Other input elements: [actionButton](actionButton), [checkboxGroupInput](checkboxGroupInput), [checkboxInput](checkboxInput), [dateInput](dateInput), [dateRangeInput](dateRangeInput), [fileInput](fileInput), [numericInput](numericInput), [passwordInput](passwordInput), [radioButtons](radioButtons), [selectInput](selectInput), [sliderInput](sliderInput), [submitButton](submitButton), [textInput](textInput), [varSelectInput](varSelectInput)

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  textAreaInput("caption", "Caption", "Data Summary", width = "1000px"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({ input$caption })
}
shinyApp(ui, server)

}
```

---

|  |  |
|---|---|
| textInput | *Create a text input control* |

---

### Description

Create an input control for entry of unstructured text values

## Usage

```
textInput(inputId, label, value = "", width = NULL,
  placeholder = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| value | Initial value. |
| width | The width of the input, e.g. '400px', or '100%'; see validateCssUnit(). |
| placeholder | A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option. |

## Value

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

## Server value

A character string of the text input. The default value is "" unless value is provided.

## See Also

updateTextInput()

Other input elements: actionButton, checkboxGroupInput, checkboxInput, dateInput, dateRangeInput, fileInput, numericInput, passwordInput, radioButtons, selectInput, sliderInput, submitButton, textAreaInput, varSelectInput

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  textInput("caption", "Caption", "Data Summary"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({ input$caption })
}
shinyApp(ui, server)
}
```

---

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 by default.

### Usage

```
textOutput(outputId, container = if (inline) span else div,
  inline = FALSE)
```

### Arguments

| | |
|---|---|
| outputId | output variable to read the value from |
| container | a function to generate an HTML element to contain the text |
| inline | use an inline (`span()`) or block container (`div()`) for the output |

### 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"))
```

---

titlePanel                    *Create a panel containing an application title.*

---

### Description

Create a panel containing an application title.

### Usage

```
titlePanel(title, windowTitle = title)
```

## Arguments

| | |
|---|---|
| `title` | An application title to display |
| `windowTitle` | The title that should be displayed by the browser window. |

## Details

Calling this function has the side effect of including a `title` tag within the head. You can also specify a page title explicitly using the `title` parameter of the top-level page function.

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  titlePanel("Hello Shiny!")
)
shinyApp(ui, server = function(input, output) { })
}
```

---

updateActionButton          *Change the label or icon of an action button on the client*

---

## Description

Change the label or icon of an action button on the client

## Usage

```
updateActionButton(session, inputId, label = NULL, icon = 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. |
| `icon` | The icon to set for the input object. To remove the current icon, use `icon=character(0)`. |

## 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.

For [radioButtons](), [checkboxGroupInput]() and [selectInput](), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

[actionButton()]()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  actionButton("update", "Update other buttons"),
  br(),
  actionButton("goButton", "Go"),
  br(),
  actionButton("goButton2", "Go 2", icon = icon("area-chart")),
  br(),
  actionButton("goButton3", "Go 3")
)

server <- function(input, output, session) {
  observe({
    req(input$update)

    # Updates goButton's label and icon
    updateActionButton(session, "goButton",
      label = "New label",
      icon = icon("calendar"))

    # Leaves goButton2's label unchaged and
    # removes its icon
    updateActionButton(session, "goButton2",
      icon = character(0))

    # Leaves goButton3's icon, if it exists,
    # unchaged and changes its label
    updateActionButton(session, "goButton3",
      label = "New label 3")
  })
}

shinyApp(ui, server)
}
```

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, inline = FALSE,
  choiceNames = NULL, choiceValues = 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 | 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. If this argument is provided, then choiceNames and choiceValues must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings. |
| selected | The values that should be initially selected, if any. |
| inline | If TRUE, render the choices inline (i.e. horizontally) |
| choiceNames | List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, choiceNames and choiceValues must have the same length). If either of these arguments is provided, then the other *must* be provided and choices *must not* be provided. The advantage of using both of these over a named list for choices is that choiceNames allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples. |
| choiceValues | List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, choiceNames and choiceValues must have the same length). If either of these arguments is provided, then the other *must* be provided and choices *must not* be provided. The advantage of using both of these over a named list for choices is that choiceNames allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples. |

## 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.

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

checkboxGroupInput()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  p("The first checkbox group controls the second"),
  checkboxGroupInput("inCheckboxGroup", "Input checkbox",
    c("Item A", "Item B", "Item C")),
  checkboxGroupInput("inCheckboxGroup2", "Input checkbox 2",
    c("Item A", "Item B", "Item C"))
)

server <- function(input, output, session) {
  observe({
    x <- input$inCheckboxGroup

    # Can use character(0) to remove all choices
    if (is.null(x))
      x <- character(0)

    # Can also set the label and select items
    updateCheckboxGroupInput(session, "inCheckboxGroup2",
      label = paste("Checkboxgroup label", length(x)),
      choices = x,
      selected = x
    )
  })
}

shinyApp(ui, server)
}
```

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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

### See Also

[checkboxInput()](#)

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 1, 0, step = 1),
  checkboxInput("inCheckbox", "Input checkbox")
)

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

```
  observe({
    # TRUE if input$controller is odd, FALSE if even.
    x_even <- input$controller %% 2 == 1

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

shinyApp(ui, server)
}
```

---

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. Supply `NA` to clear the date. |
| 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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

**See Also**

[dateInput()](dateInput())

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("n", "Day of month", 1, 30, 10),
  dateInput("inDate", "Input date")
)

server <- function(input, output, session) {
  observe({
    date <- as.Date(paste0("2013-04-", input$n))
    updateDateInput(session, "inDate",
      label = paste("Date label", input$n),
      value = date,
      min   = date - 3,
      max   = date + 3
    )
  })
}

shinyApp(ui, server)
}
```

---

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. Supplying NA clears the start date. |

| | |
|---|---|
| end | The end date. Either a Date object, or a string in `yyyy-mm-dd` format. Supplying NA clears the end date. |
| 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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

[dateRangeInput()](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("n", "Day of month", 1, 30, 10),
  dateRangeInput("inDateRange", "Input date range")
)

server <- function(input, output, session) {
  observe({
    date <- as.Date(paste0("2013-04-", input$n))

    updateDateRangeInput(session, "inDateRange",
      label = paste("Date range label", input$n),
      start = date - 1,
      end = date + 1,
      min = date - 5,
      max = date + 5
    )
  })
}

shinyApp(ui, server)
}
```

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.

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

### See Also

numericInput()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 20, 10),
  numericInput("inNumber", "Input number", 0),
  numericInput("inNumber2", "Input number 2", 0)
)

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

  observeEvent(input$controller, {
    # We'll use the input$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)
  })
}

shinyApp(ui, server)
}
```

---

updateQueryString        *Update URL in browser's location bar*

---

### Description

This function updates the client browser's query string in the location bar. It typically is called from an observer. Note that this will not work in Internet Explorer 9 and below.

### Usage

```
updateQueryString(queryString, mode = c("replace", "push"),
  session = getDefaultReactiveDomain())
```

### Arguments

| | |
|---|---|
| queryString | The new query string to show in the location bar. |
| mode | When the query string is updated, should the the current history entry be replaced (default), or should a new history entry be pushed onto the history stack? The former should only be used in a live bookmarking context. The latter is useful if you want to navigate between states using the browser's back and forward buttons. See Examples. |
| session | A Shiny session object. |

**Details**

For mode = "push", only three updates are currently allowed:

1. the query string (format: ?param1=val1&param2=val2)

2. the hash (format: #hash)

3. both the query string and the hash (format: ?param1=val1&param2=val2#hash)

In other words, if mode = "push", the queryString must start with either ? or with #.

A technical curiosity: under the hood, this function is calling the HTML5 history API (which is where the names for the mode argument come from). When mode = "replace", the function called is window.history.replaceState(null,null,queryString). When mode = "push", the function called is window.history.pushState(null,null,queryString).

**See Also**

enableBookmarking(), getQueryString()

**Examples**

```
## Only run these examples in interactive sessions
if (interactive()) {

  ## App 1: Doing "live" bookmarking
  ## Update the browser's location bar every time an input changes.
  ## This should not be used with enableBookmarking("server"),
  ## because that would create a new saved state on disk every time
  ## the user changes an input.
  enableBookmarking("url")
  shinyApp(
    ui = function(req) {
      fluidPage(
        textInput("txt", "Text"),
        checkboxInput("chk", "Checkbox")
      )
    },
    server = function(input, output, session) {
      observe({
        # Trigger this observer every time an input changes
        reactiveValuesToList(input)
        session$doBookmark()
      })
      onBookmarked(function(url) {
        updateQueryString(url)
      })
    }
  )

  ## App 2: Printing the value of the query string
  ## (Use the back and forward buttons to see how the browser
  ## keeps a record of each state)
  shinyApp(
```

```
      ui = fluidPage(
        textInput("txt", "Enter new query string"),
        helpText("Format: ?param1=val1&param2=val2"),
        actionButton("go", "Update"),
        hr(),
        verbatimTextOutput("query")
      ),
      server = function(input, output, session) {
        observeEvent(input$go, {
          updateQueryString(input$txt, mode = "push")
        })
        output$query <- renderText({
          query <- getQueryString()
          queryText <- paste(names(query), query,
                        sep = "=", collapse=", ")
          paste("Your query string is:\n", queryText)
        })
      }
    )
  }
```

---

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, inline = FALSE, choiceNames = NULL,
  choiceValues = 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 | 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). If this argument is provided, then `choiceNames` and `choiceValues` must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings. |
| selected | The initially selected value (if not specified then defaults to the first value) |
| inline | If TRUE, render the choices inline (i.e. horizontally) |

choiceNames          List of names and values, respectively, that are displayed to the user in the app
                     and correspond to the each choice (for this reason, choiceNames and choiceValues
                     must have the same length). If either of these arguments is provided, then the
                     other *must* be provided and choices *must not* be provided. The advantage of
                     using both of these over a named list for choices is that choiceNames allows
                     any type of UI object to be passed through (tag objects, icons, HTML code, ...),
                     instead of just simple text. See Examples.

choiceValues         List of names and values, respectively, that are displayed to the user in the app
                     and correspond to the each choice (for this reason, choiceNames and choiceValues
                     must have the same length). If either of these arguments is provided, then the
                     other *must* be provided and choices *must not* be provided. The advantage of
                     using both of these over a named list for choices is that choiceNames allows
                     any type of UI object to be passed through (tag objects, icons, HTML code, ...),
                     instead of just simple text. See Examples.

## 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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be
cleared by using choices=character(0). Similarly, for these inputs, the selected item can be
cleared by using selected=character(0).

## See Also

[radioButtons()](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  p("The first radio button group controls the second"),
  radioButtons("inRadioButtons", "Input radio buttons",
    c("Item A", "Item B", "Item C")),
  radioButtons("inRadioButtons2", "Input radio buttons 2",
    c("Item A", "Item B", "Item C"))
)

server <- function(input, output, session) {
  observe({
    x <- input$inRadioButtons
```

```
    # Can also set the label and select items
    updateRadioButtons(session, "inRadioButtons2",
      label = paste("radioButtons label", x),
      choices = x,
      selected = x
    )
  })
}

shinyApp(ui, server)
}
```

---

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)

updateSelectizeInput(session, inputId, label = NULL, choices = NULL,
  selected = NULL, options = list(), server = FALSE)

updateVarSelectInput(session, inputId, label = NULL, data = NULL,
  selected = NULL)

updateVarSelectizeInput(session, inputId, label = NULL, data = NULL,
  selected = NULL, options = list(), server = FALSE)
```

### 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 | 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. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the group labels (leveraging the `<optgroup>` HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature. |
| selected | The initially selected value (or multiple values if `multiple = TRUE`). If not specified then defaults to the first value for single-select lists and no values for multiple select lists. |

| | |
|---|---|
| options | A list of options. See the documentation of **selectize.js** for possible options (character option values inside base::I() will be treated as literal JavaScript code; see renderDataTable() for details). |
| server | whether to store choices on the server side, and load the select options dynamically on searching, instead of writing all choices into the page at once (i.e., only use the client-side version of **selectize.js**) |
| data | A data frame. Used to retrieve the column names as choices for a selectInput() |

## 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.

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

selectInput() varSelectInput()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  p("The checkbox group controls the select input"),
  checkboxGroupInput("inCheckboxGroup", "Input checkbox",
    c("Item A", "Item B", "Item C")),
  selectInput("inSelect", "Select input",
    c("Item A", "Item B", "Item C"))
)

server <- function(input, output, session) {
  observe({
    x <- input$inCheckboxGroup

    # Can use character(0) to remove all choices
    if (is.null(x))
      x <- character(0)

    # Can also set the label and select items
    updateSelectInput(session, "inSelect",
      label = paste("Select input label", length(x)),
```

```
      choices = x,
      selected = tail(x, 1)
    )
  })
}

shinyApp(ui, server)
}
```

---

updateSliderInput          *Update Slider Input Widget*

---

#### Description

Change the value of a slider input on the client.

#### Usage

```
updateSliderInput(session, inputId, label = NULL, value = NULL,
  min = NULL, max = NULL, step = NULL, timeFormat = NULL,
  timezone = 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. |
| timeFormat | Date and POSIXt formatting. |
| timezone | The timezone offset for POSIXt objects. |

#### 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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

[sliderInput()](#)

## Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(
          p("The first slider controls the second"),
          sliderInput("control", "Controller:", min=0, max=20, value=10,
                      step=1),
          sliderInput("receive", "Receiver:", min=0, max=20, value=10,
                      step=1)
        ),
        mainPanel()
      )
    ),
    server = function(input, output, session) {
      observe({
        val <- input$control
        # Control the value, min, max, and step.
        # Step size is 2 when input value is even; 1 when value is odd.
        updateSliderInput(session, "receive", value = val,
          min = floor(val/2), max = val+4, step = (val+1)%%2 + 1)
      })
    }
  )
}
```

updateTabsetPanel          *Change the selected tab on the client*

## Description

Change the selected tab on the client

## Usage

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

updateNavbarPage(session, inputId, selected = NULL)

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

## Arguments

| | |
|---|---|
| session | The session object passed to function given to shinyServer. |
| inputId | The id of the tabsetPanel, navlistPanel, or navbarPage object. |
| selected | The name of the tab to make active. |

## See Also

[tabsetPanel()](), [navlistPanel()](), [navbarPage()]()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(sidebarLayout(
  sidebarPanel(
    sliderInput("controller", "Controller", 1, 3, 1)
  ),
  mainPanel(
    tabsetPanel(id = "inTabset",
      tabPanel(title = "Panel 1", value = "panel1", "Panel 1 content"),
      tabPanel(title = "Panel 2", value = "panel2", "Panel 2 content"),
      tabPanel(title = "Panel 3", value = "panel3", "Panel 3 content")
    )
  )
))

server <- function(input, output, session) {
  observeEvent(input$controller, {
    updateTabsetPanel(session, "inTabset",
      selected = paste0("panel", input$controller)
    )
  })
}

shinyApp(ui, server)
}
```

---

updateTextAreaInput    *Change the value of a textarea input on the client*

---

## Description

Change the value of a textarea input on the client

## Usage

```
updateTextAreaInput(session, inputId, label = NULL, value = NULL,
  placeholder = 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. |
| placeholder | The placeholder 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.

For [radioButtons](#)(), [checkboxGroupInput](#)() and [selectInput](#)(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

## See Also

[textAreaInput()](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 20, 10),
  textAreaInput("inText", "Input textarea"),
  textAreaInput("inText2", "Input textarea 2")
)

server <- 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
    updateTextAreaInput(session, "inText", value = paste("New text", x))

    # Can also set the label, this time for input$inText2
    updateTextAreaInput(session, "inText2",
      label = paste("New label", x),
      value = paste("New text", x))
```

```
  })
}

shinyApp(ui, server)
}
```

---

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,
  placeholder = 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. |
| placeholder | The placeholder 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.

For [radioButtons](), (), [checkboxGroupInput]() and [selectInput](), (), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

### See Also

[textInput()]()

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 20, 10),
  textInput("inText", "Input text"),
  textInput("inText2", "Input text 2")
)

server <- 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))
  })
}

shinyApp(ui, server)
}
```

---

urlModal                    *Generate a modal dialog that displays a URL*

---

## Description

The modal dialog generated by urlModal will display the URL in a textarea input, and the URL text will be selected so that it can be easily copied. The result from urlModal should be passed to the [showModal()](#) function to display it in the browser.

## Usage

```
urlModal(url, title = "Bookmarked application link", subtitle = NULL)
```

## Arguments

| | |
|---|---|
| url | A URL to display in the dialog box. |
| title | A title for the dialog box. |
| subtitle | Text to display underneath URL. |

## validate            *Validate input values and other conditions*

### Description

For an output rendering function (e.g. [renderPlot()](#)), you may need to check that certain input values are available and valid before you can render the output. validate gives you a convenient mechanism for doing so.

### Usage

```
validate(..., errorClass = character(0))

need(expr, message = paste(label, "must be provided"), label)
```

### Arguments

| | |
|---|---|
| ... | A list of tests. Each test should equal NULL for success, FALSE for silent failure, or a string for failure with an error message. |
| errorClass | A CSS class to apply. The actual CSS string will have shiny-output-error-prepended to this value. |
| expr | An expression to test. The condition will pass if the expression meets the conditions spelled out in Details. |
| message | A message to convey to the user if the validation condition is not met. If no message is provided, one will be created using label. To fail with no message, use FALSE for the message. |
| label | A human-readable name for the field that may be missing. This parameter is not needed if message is provided, but must be provided otherwise. |

### Details

The validate function takes any number of (unnamed) arguments, each of which represents a condition to test. If any of the conditions represent failure, then a special type of error is signaled which stops execution. If this error is not handled by application-specific code, it is displayed to the user by Shiny.

An easy way to provide arguments to validate is to use the need function, which takes an expression and a string; if the expression is considered a failure, then the string will be used as the error message. The need function considers its expression to be a failure if it is any of the following:

- FALSE
- NULL
- ""
- An empty atomic vector
- An atomic vector that contains only missing values

- A logical vector that contains all FALSE or missing values
- An object of class "try-error"
- A value that represents an unclicked actionButton()

If any of these values happen to be valid, you can explicitly turn them to logical values. For example, if you allow NA but not NULL, you can use the condition !is.null(input$foo), because !is.null(NA) == TRUE.

If you need validation logic that differs significantly from need, you can create other validation test functions. A passing test should return NULL. A failing test should return an error message as a single-element character vector, or if the failure should happen silently, FALSE.

Because validation failure is signaled as an error, you can use validate in reactive expressions, and validation failures will automatically propagate to outputs that use the reactive expression. In other words, if reactive expression a needs input$x, and two outputs use a (and thus depend indirectly on input$x), it's not necessary for the outputs to validate input$x explicitly, as long as a does validate it.

### Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

ui <- fluidPage(
  checkboxGroupInput('in1', 'Check some letters', choices = head(LETTERS)),
  selectizeInput('in2', 'Select a state', choices = state.name),
  plotOutput('plot')
)

server <- function(input, output) {
  output$plot <- renderPlot({
    validate(
      need(input$in1, 'Check at least one letter!'),
      need(input$in2 != '', 'Please choose a state.')
    )
    plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))
  })
}

shinyApp(ui, server)

}
```

---

| validateCssUnit | *Validate proper CSS formatting of a unit* |
|---|---|

---

### Description

Checks that the argument is valid for use as a CSS unit of length.

## Usage

```
validateCssUnit(x)
```

## Arguments

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

## Details

NULL and NA are returned unchanged.

Single element numeric vectors are returned as a character vector with the number plus a suffix of
"px".

Single element character vectors must be "auto" or "inherit", a number, or a length calculated
by the "calc" CSS function. If the number has a suffix, it must be valid: px, %, ch, em, rem, pt, in,
cm, mm, ex, pc, vh, vw, vmin, or vmax. If the number has no suffix, the suffix "px" is appended.

Any other value will cause an error to be thrown.

## Value

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

## Examples

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

---

varSelectInput                     *Select variables from a data frame*

---

## Description

Create a select list that can be used to choose a single or multiple items from the column names of
a data frame.

## Usage

```
varSelectInput(inputId, label, data, selected = NULL, multiple = FALSE,
  selectize = TRUE, width = NULL, size = NULL)

varSelectizeInput(inputId, ..., options = NULL, width = NULL)
```

## Arguments

| | |
|---|---|
| inputId | The input slot that will be used to access the value. |
| label | Display label for the control, or NULL for no label. |
| data | A data frame. Used to retrieve the column names as choices for a [selectInput()](#) |
| selected | The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists. |
| multiple | Is selection of multiple items allowed? |
| selectize | Whether to use **selectize.js** or not. |
| width | The width of the input, e.g. '400px', or '100%'; see [validateCssUnit()](#). |
| size | Number of items to show in the selection box; a larger number will result in a taller box. Not compatible with selectize=TRUE. Normally, when multiple=FALSE, a select input will be a drop-down list, but when size is set, it will be a box instead. |
| ... | Arguments passed to varSelectInput(). |
| options | A list of options. See the documentation of **selectize.js** for possible options (character option values inside [base::I()](#) will be treated as literal JavaScript code; see [renderDataTable()](#) for details). |

## Details

By default, varSelectInput() and selectizeInput() use the JavaScript library **selectize.js** ([https://github.com/selectize/selectize.js](https://github.com/selectize/selectize.js)) to instead of the basic select input element. To use the standard HTML select input element, use selectInput() with selectize=FALSE.

## Value

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

## Server value

The resulting server input value will be returned as:

- A symbol if multiple = FALSE. The input value should be used with rlang's [rlang::!!()](#). For example, ggplot2::aes(!!input$variable).
- A list of symbols if multiple = TRUE. The input value should be used with rlang's [rlang::!!!()](#) to expand the symbol list as individual arguments. For example, dplyr::select(mtcars,!!!input$variabls) which is equivalent to dplyr::select(mtcars,!!input$variabls[[1]],!!input$variabls[[2]],...,!!input$

## Note

The variable selectize input created from varSelectizeInput() allows deletion of the selected option even in a single select input, which will return an empty string as its value. This is the default behavior of **selectize.js**. However, the selectize input created from selectInput(...,selectize = TRUE) will ignore the empty string value when it is a single choice input and the empty string is not in the choices argument. This is to keep compatibility with selectInput(...,selectize = FALSE).

**See Also**

[updateSelectInput()](updateSelectInput())

Other input elements: [actionButton](actionButton), [checkboxGroupInput](checkboxGroupInput), [checkboxInput](checkboxInput), [dateInput](dateInput), [dateRangeInput](dateRangeInput), [fileInput](fileInput), [numericInput](numericInput), [passwordInput](passwordInput), [radioButtons](radioButtons), [selectInput](selectInput), [sliderInput](sliderInput), [submitButton](submitButton), [textAreaInput](textAreaInput), [textInput](textInput)

**Examples**

```
## Only run examples in interactive R sessions
if (interactive()) {

library(ggplot2)

# single selection
shinyApp(
  ui = fluidPage(
    varSelectInput("variable", "Variable:", mtcars),
    plotOutput("data")
  ),
  server = function(input, output) {
    output$data <- renderPlot({
      ggplot(mtcars, aes(!!input$variable)) + geom_histogram()
    })
  }
)


# multiple selections
## Not run:
shinyApp(
 ui = fluidPage(
   varSelectInput("variables", "Variable:", mtcars, multiple = TRUE),
   tableOutput("data")
 ),
 server = function(input, output) {
   output$data <- renderTable({
       if (length(input$variables) == 0) return(mtcars)
       mtcars %>% dplyr::select(!!!input$variables)
   }, rownames = TRUE)
 }
)
## End(Not run)

}
```

---

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, placeholder = FALSE)
```

**Arguments**

| | |
|---|---|
| outputId | output variable to read the value from |
| placeholder | if the output is empty or NULL, should an empty rectangle be displayed to serve as a placeholder? (does not affect behavior when the the output in nonempty) |

**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**

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = basicPage(
      textInput("txt", "Enter the text to display below:"),
      verbatimTextOutput("default"),
      verbatimTextOutput("placeholder", placeholder = TRUE)
    ),
    server = function(input, output) {
      output$default <- renderText({ input$txt })
      output$placeholder <- renderText({ input$txt })
    }
  )
}
```

---

verticalLayout                  *Lay out UI elements vertically*

---

**Description**

Create a container that includes one or more rows of content (each element passed to the container
will appear on it's own line in the UI)

## Usage

```
verticalLayout(..., fluid = TRUE)
```

## Arguments

| | |
|---|---|
| ... | Elements to include within the container |
| fluid | TRUE to use fluid layout; FALSE to use fixed layout. |

## See Also

Other layout functions: fillPage, fixedPage, flowLayout, fluidPage, navbarPage, sidebarLayout, splitLayout

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  verticalLayout(
    a(href="http://example.com/link1", "Link One"),
    a(href="http://example.com/link2", "Link Two"),
    a(href="http://example.com/link3", "Link Three")
  )
)
shinyApp(ui, server = function(input, output) { })
}
```

---

viewer                              *Viewer options*

---

## Description

Use these functions to control where the gadget is displayed in RStudio (or other R environments that emulate RStudio's viewer pane/dialog APIs). If viewer APIs are not available in the current R environment, then the gadget will be displayed in the system's default web browser (see utils::browseURL()).

## Usage

```
paneViewer(minHeight = NULL)

dialogViewer(dialogName, width = 600, height = 600)

browserViewer(browser = getOption("browser"))
```

## Arguments

| | |
|---|---|
| `minHeight` | The minimum height (in pixels) desired to show the gadget in the viewer pane. If a positive number, resize the pane if necessary to show at least that many pixels. If NULL, use the existing viewer pane size. If "maximize", use the maximum available vertical space. |
| `dialogName` | The window title to display for the dialog. |
| `width, height` | The desired dialog width/height, in pixels. |
| `browser` | See [utils::browseURL()](). |

## Value

A function that takes a single `url` parameter, suitable for passing as the `viewer` argument of [runGadget()]().

---

wellPanel                          *Create a well panel*

---

## Description

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

## Usage

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

## Arguments

| | |
|---|---|
| `...` | UI elements to include inside the panel. |

## Value

The newly created panel.

---

withMathJax                    *Load the MathJax library and typeset math expressions*

---

### Description

This function adds MathJax to the page and typeset the math expressions (if found) in the content
.... It only needs to be called once in an app unless the content is rendered *after* the page is
loaded, e.g. via renderUI(), in which case we have to call it explicitly every time we write math
expressions to the output.

### Usage

```
withMathJax(...)
```

### Arguments

...          any HTML elements to apply MathJax to

### Examples

```
withMathJax(helpText("Some math here $$\\alpha+\\beta$$"))
# now we can just write "static" content without withMathJax()
div("more math here $$\\sqrt{2}$$")
```

---

withProgress                    *Reporting progress (functional API)*

---

### Description

Reports progress to the user during long-running operations.

### Usage

```
withProgress(expr, min = 0, max = 1, value = min + (max - min) * 0.1,
  message = NULL, detail = NULL,
  style = getShinyOption("progress.style", default = "notification"),
  session = getDefaultReactiveDomain(), env = parent.frame(),
  quoted = FALSE)

setProgress(value = NULL, message = NULL, detail = NULL,
  session = getDefaultReactiveDomain())

incProgress(amount = 0.1, message = NULL, detail = NULL,
  session = getDefaultReactiveDomain())
```

## Arguments

| | |
|---|---|
| expr | The work to be done. This expression should contain calls to setProgress. |
| min | The value that represents the starting point of the progress bar. Must be less tham max. Default is 0. |
| max | The value that represents the end of the progress bar. Must be greater than min. Default is 1. |
| value | Single-element numeric vector; the value at which to set the progress bar, relative to min and max. |
| message | A single-element character vector; the message to be displayed to the user, or NULL to hide the current message (if any). |
| detail | A single-element character vector; the detail message to be displayed to the user, or NULL to hide the current detail message (if any). The detail message will be shown with a de-emphasized appearance relative to message. |
| style | Progress display style. If "notification" (the default), the progress indicator will show using Shiny's notification API. If "old", use the same HTML and CSS used in Shiny 0.13.2 and below (this is for backward-compatibility). |
| session | The Shiny session object, as provided by shinyServer to the server function. The default is to automatically find the session by using the current reactive domain. |
| env | The environment in which expr should be evaluated. |
| quoted | Whether expr is a quoted expression (this is not common). |
| amount | For incProgress, the amount to increment the status bar. Default is 0.1. |

## Details

This package exposes two distinct programming APIs for working with progress. Using withProgress with incProgress or setProgress provide a simple function-based interface, while the [Progress()](#) reference class provides an object-oriented API.

Use withProgress to wrap the scope of your work; doing so will cause a new progress panel to be created, and it will be displayed the first time incProgress or setProgress are called. When withProgress exits, the corresponding progress panel will be removed.

The incProgress function increments the status bar by a specified amount, whereas the setProgress function sets it to a specific value, and can also set the text displayed.

Generally, withProgress/incProgress/setProgress should be sufficient; the exception is if the work to be done is asynchronous (this is not common) or otherwise cannot be encapsulated by a single scope. In that case, you can use the Progress reference class.

As of version 0.14, the progress indicators use Shiny's new notification API. If you want to use the old styling (for example, you may have used customized CSS), you can use style="old" each time you call withProgress(). If you don't want to set the style each time withProgress is called, you can instead call [shinyOptions(progress.style="old")](#) just once, inside the server function.

## See Also

[Progress()](#)

## Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
options(device.ask.default = FALSE)

ui <- fluidPage(
  plotOutput("plot")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    withProgress(message = 'Calculation in progress',
                 detail = 'This may take a while...', value = 0, {
      for (i in 1:15) {
        incProgress(1/15)
        Sys.sleep(0.25)
      }
    })
    plot(cars)
  })
}

shinyApp(ui, server)
}
```

---

| withTags | *Evaluate an expression using* 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