Package ‘shiny’

February 20, 2015

Type Package

Title Web Application Framework for R

Version 0.11.1

Date 2015-02-10

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

License GPL-3 | file LICENSE

Depends R (>= 3.0.0)

Imports tools, utils, httpuv (>= 1.3.2), mime (>= 0.1.3), RJSONIO, xtable, digest, htmltools (>= 0.2.6), R6 (>= 2.0)

Suggests datasets, Cairo (>= 1.5-5), testthat, knitr (>= 1.6), markdown

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

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

Collate 'app.R' 'bootstrap-layout.R' 'map.R' 'globals.R' 'utils.R' 'bootstrap.R' 'cache.R' 'fileupload.R' 'stack.R' 'graph.R' 'hooks.R' 'html-deps.R' 'htmltools.R' 'imageutils.R' 'jqueryui.R' 'middleware-shiny.R' 'middleware.R' 'priorityqueue.R' 'progress.R' 'react.R' 'reactive-domains.R' 'reactives.R' 'run-url.R' 'server.R' 'shiny.R' 'shinyui.R' 'shinywrappers.R' 'showcase.R' 'slider.R' 'tar.R' 'timer.R' 'update-input.R'

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/1.10.4/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),
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>

NeedsCompilation no

Repository CRAN

Date/Publication 2015-02-11 01:00:37

R topics documented:

shiny-package 5
absolutePanel 5
actionButton 7
addResourcePath 8
bootstrapPage 8
builder 9
checkboxGroupInput 11
checkboxInput 12
column 12
conditionalPanel 13
createWebDependency 15
dateInput 15
dateRangeInput 17
domains 19
downloadButton 20
downloadHandler 21
exprToFunction 22
fileInput 23

fixedPage . . . 24
flowLayout . . . 25
fluidPage . . . 26
headerPanel . . . 27
helpText . . . 28
HTML . . . 29
htmlOutput . . . 29
icon . . . 30
imageOutput . . . 31
include . . . 32
inputPanel . . . 33
installExprFunction . . . 33
invalidateLater . . . 34
is.reactivevalues . . . 35
isolate . . . 35
knitr_methods . . . 37
knit_print.html . . . 37
mainPanel . . . 38
makeReactiveBinding . . . 38
markRenderFunction . . . 39
maskReactiveContext . . . 40
navbarPage . . . 40
navlistPanel . . . 42
numericInput . . . 43
observe . . . 44
observeEvent . . . 45
outputOptions . . . 48
pageWithSidebar . . . 49
parseQueryString . . . 50
passwordInput . . . 51
plotOutput . . . 52
plotPNG . . . 53
Progress . . . 54
radioButtons . . . 55
reactive . . . 56
reactiveFileReader . . . 57
reactivePlot . . . 59
reactivePoll . . . 59
reactivePrint . . . 60
reactiveTable . . . 61
reactiveText . . . 61
reactiveTimer . . . 62
reactiveUI . . . 63
reactiveValues . . . 63
reactiveValuesToList . . . 64
registerInputHandler . . . 65
removeInputHandler . . . 66
renderDataTable . . . 67

renderImage . . . 68
renderPlot . . . 70
renderPrint . . . 71
renderTable . . . 73
renderText . . . 74
renderUI . . . 75
repeatable . . . 76
runApp . . . 77
runExample . . . 79
runUrl . . . 80
selectInput . . . 81
serverInfo . . . 83
session . . . 83
shiny-options . . . 85
shinyApp . . . 86
shinyServer . . . 88
shinyUI . . . 89
showReactLog . . . 89
sidebarLayout . . . 90
sidebarPanel . . . 91
singleton . . . 92
sliderInput . . . 92
splitLayout . . . 94
stopApp . . . 95
submitButton . . . 95
tableOutput . . . 96
tabPanel . . . 97
tabsetPanel . . . 98
tag . . . 99
textInput . . . 100
textOutput . . . 101
titlePanel . . . 101
updateCheckboxGroupInput . . . 102
updateCheckboxInput . . . 103
updateDateInput . . . 104
updateDateRangeInput . . . 105
updateNumericInput . . . 107
updateRadioButtons . . . 108
updateSelectInput . . . 109
updateSliderInput . . . 111
updateTabsetPanel . . . 112
updateTextInput . . . 113
validate . . . 114
validateCssUnit . . . 116
verbatimTextOutput . . . 117
verticalLayout . . . 117
wellPanel . . . 118
withMathJax . . . 119

withProgress 119
withTags 121

Index **122**

`shiny-package` *Web Application Framework for R*

Description

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

Details

The Shiny tutorial at `http://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, ...)

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

Arguments

`inputId`	Specifies 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.
`...`	Named attributes to be applied to the button or link.

See Also

`observeEvent` and `eventReactive`

Other input.elements: `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

```
## Not run:
# In server.R
output$distPlot <- renderPlot({
  # Take a dependency on input$goButton
  input$goButton

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

# In ui.R
actionButton("goButton", "Go!")

## End(Not run)
```

`addResourcePath` *Resource Publishing*

Description

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

Usage

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

Arguments

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

Details

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

See Also

`singleton`

Examples

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

`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`

`builder` *HTML Builder Functions*

Description

Simple functions for constructing HTML documents.

Usage

```
tags

p(...)

h1(...)

h2(...)

h3(...)

h4(...)
```

```
h5(...)

h6(...)

a(...)

br(...)

div(...)

span(...)

pre(...)

code(...)

img(...)

strong(...)

em(...)

hr(...)
```

Arguments

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

Details

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

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

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

Examples

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

`checkboxGroupInput` *Checkbox Group Input Control*

Description

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

Usage

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

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.
`selected`	The values that should be initially selected, if any.
`inline`	If `TRUE`, render the choices inline (i.e. horizontally)

Value

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

See Also

`checkboxInput`, `updateCheckboxGroupInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

checkboxInput *Checkbox Input Control*

Description

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

Usage

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

Arguments

`inputId`	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`).

Value

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

See Also

`checkboxGroupInput`, `updateCheckboxInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

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

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

fluidRow(
  column(width = 4,
    "4"
  ),
  column(width = 3, offset = 2,
    "3 offset 2"
  )
)
```

`conditionalPanel` *Conditional Panel*

Description

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

Usage

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

Arguments

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

Details

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

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

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

   # Only show this panel if the plot type is a histogram
   conditionalPanel(
      condition = "input.plotType == 'hist'",
      selectInput(
         "breaks", "Breaks",
         c("Sturges",
           "Scott",
           "Freedman-Diaconis",
           "[Custom]" = "custom")),

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

`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` to map the HTML dependency's directory to a URL.

Usage

```
createWebDependency(dependency)
```

Arguments

`dependency`	A single HTML dependency object, created using `htmlDependency`. If the `src` value is named, then `href` and/or `file` names must be present.

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")
```

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 "bg", "ca", "cs", "da", "de", "el", "es", "fi", "fr", "he", "hr", "hu", "id", "is", "it", "ja", "kr", "lt", "lv", "ms", "nb", "nl", "pl", "pt", "pt-BR", "ro", "rs", "rs-latin", "ru", "sk", "sl", "sv", "sw", "th", "tr", "uk", "zh-CN", and "zh-TW".

Details

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

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

See Also

`dateRangeInput`, `updateDateInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

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

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

# Pass in a Date object
```

```
dateInput("date", "Date:", value = Sys.Date()-10)

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

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

dateRangeInput *Create date range input*

Description

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

Usage

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

Arguments

inputId	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 "bg", "ca", "cs", "da", "de", "el", "es", "fi", "fr", "he", "hr", "hu", "id", "is", "it", "ja", "kr", "lt", "lv", "ms", "nb", "nl", "pl", "pt", "pt-BR", "ro", "rs", "rs-latin", "ru", "sk", "sl", "sv", "sw", "th", "tr", "uk", "zh-CN", and "zh-TW".
separator	String to display between the start and end input boxes.

Details

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

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

See Also

`dateInput`, `updateDateRangeInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

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

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

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

# Use different language and different first day of week
```

```
dateRangeInput("daterange", "Date range:",
               language = "de",
               weekstart = 1)

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

`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.

See Also

downloadHandler

Examples

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

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

## End(Not run)
```

downloadHandler *File Downloads*

Description

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

Usage

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

Arguments

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

Examples

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

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

## End(Not run)
```

exprToFunction *Convert an expression to a function*

Description

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

Usage

```
exprToFunction(expr, env = parent.frame(2), quoted = FALSE,
  caller_offset = 1)
```

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?
`caller_offset`	If specified, the offset in the callstack of the functiont to be treated as the caller.

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)
```

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.

Details

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

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

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

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

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

See Also

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

`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` within it. Note that unlike `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 function.

Note

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

See Also

column

Examples

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

`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‘ 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

verticalLayout

Examples

```
flowLayout(
  numericInput("rows", "How many rows?", 5),
  selectInput("letter", "Which letter?", LETTERS),
  sliderInput("value", "What value?", 0, 100, 50)
)
```

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`, `sidebarLayout`

Examples

```
shinyUI(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")
    )
  )
))

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

`headerPanel` *Create a header panel*

Description

Create a header panel containing an application title.

Usage

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

Arguments

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

Value

A headerPanel that can be passed to pageWithSidebar

Examples

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

`helpText` *Create a help text element*

Description

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

Usage

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

Arguments

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

Value

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

Examples

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

HTML *Mark Characters as HTML*

Description

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

Usage

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

Arguments

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

Value

The same value, but marked as HTML.

Examples

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

htmlOutput *Create an HTML output element*

Description

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

Usage

```
htmlOutput(outputId, 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")
)
```

`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 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

```
icon("calendar")               # standard icon
icon("calendar", "fa-3x")      # 3x normal size
icon("cog", lib = "glyphicon") # From glyphicon library

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

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

`imageOutput` *Create a image output element*

Description

Render a renderImage within an application page.

Usage

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

Arguments

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

Value

An image output element that can be included in a panel

Examples

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

---

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.

`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 = as.character(sys.call(-1)[[1]]))
```

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.

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)
```

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

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

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

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

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

  ## End(Not run)
```

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

Description

Checks whether its argument is a reactivevalues object.

Usage

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

Arguments

`x`	The object to test.

See Also

`reactiveValues`.

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

Description

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

Usage

```
isolate(expr)
```

Arguments

`expr`	An expression that can access reactive values or expressions.

Details

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

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

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

Examples

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

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

observe({
  input$saveButton  # Do take a dependency on input$saveButton

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

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

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


## End(Not run)

# Can also use isolate to call reactive expressions from the R console
values <- reactiveValues(A=1)
```

```
fun <- reactive({ as.character(values$A) })
isolate(fun())
# "1"

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

`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)
```

Arguments

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

`knit_print.html` *Knitr S3 methods*

Description

These S3 methods are necessary to allow HTML tags to print themselves in knitr/rmarkdown documents.

Usage

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

knit_print.html(x, ...)

knit_print.shiny.tag.list(x, ...)
```

Arguments

`x`	Object to knit_print
`...`	Additional knit_print arguments

`mainPanel` *Create a main panel*

Description

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

Usage

```
mainPanel(..., width = 8)
```

Arguments

`...`	Output elements to include in the main panel
`width`	The width of the main panel. For fluid layouts this is out of 12 total units; for fixed layouts it is out of whatever the width of the main panel's parent column is.

Value

A main panel that can be passed to `sidebarLayout`.

Examples

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

`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)
```

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.

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`

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` elements.

Usage

```
navbarPage(title, ..., id = 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, ..., icon = NULL)
```

Arguments

`title`	The title to display in the navbar
`...`	`tabPanel` elements to include in the page
`id`	If provided, you can use `input$id` in your server logic to determine which of the current tabs is active. The value will correspond to the `value` argument that is passed to `tabPanel`.
`position`	Determines whether the navbar should be displayed at the top of the page with normal scrolling behavior (`"static-top"`), pinned at the top (`"fixed-top"`), or pinned at the bottom (`"fixed-bottom"`). Note that using `"fixed-top"` or `"fixed-bottom"` will cause the navbar to overlay your body content, unless you add padding, e.g.: `tags$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 touchscreen 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.
`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`

Examples

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

shinyUI(navbarPage("App Title",
  tabPanel("Plot"),
  navbarMenu("More",
    tabPanel("Summary"),
    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.

Examples

```
shinyUI(fluidPage(

  titlePanel("Application Title"),

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

`numericInput` *Create a numeric input control*

Description

Create an input control for entry of numeric values

Usage

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

Arguments

`inputId`	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

Value

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

See Also

`updateNumericInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

observe — *Create a reactive observer*

Description

Creates an observer from the given expression.

Usage

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

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)

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

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 executed 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.
`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.

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).

Value

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

See Also

`actionButton`

Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  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$button, function() {
      cat("Showing", input$x, "rows\n")
    })
    # Take a reactive dependency on input$button, but
    # not on any of the stuff inside the function
    df <- eventReactive(input$button, function() {
      head(cars, input$x)
    })
    output$table <- renderTable({
      df()
    })
  }
  shinyApp(ui=ui, server=server)
}
```

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

Description

These are the available options for an output object:

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

Usage

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

Arguments

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

Examples

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

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

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

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

## End(Not run)
```

`pageWithSidebar` *Create a page with a sidebar*

Description

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

Usage

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

Arguments

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

Value

A UI defintion that can be passed to the shinyUI function

Note

This function is deprecated. You should use `fluidPage` along with `sidebarLayout` to implement a page with a sidebar.

Examples

```
# Define UI
shinyUI(pageWithSidebar(

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

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

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

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

Description

Returns a named 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 = lis` `'z')))` when `nested = TRUE`, and ``list(`a[i1][j1]` = 'x',`b[i1][j1]` = 'y', `b[i2][j1]` = '`` when `nested = FALSE`.

Examples

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

## Not run:
# Example of usage within a Shiny app
shinyServer(function(input, output, clientData) {
  output$queryText <- renderText({
    query <- parseQueryString(clientData$url_search)

    # Ways of accessing the values
    if (as.numeric(query$foo) == 1) {
      # Do something
    }
    if (query[["bar"]] == "targetstring") {
      # Do something else
    }

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

## End(Not run)
```

passwordInput *Create a password input control*

Description

Create an password control for entry of passwords.

Usage

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

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.

Value

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

See Also

updateTextInput

Other input.elements: actionButton, actionLink; animationOptions, sliderInput; checkboxGroupInput; checkboxInput; dateInput; dateRangeInput; fileInput; numericInput; radioButtons; selectInput, selectizeInput; submitButton; textInput

Examples

```
passwordInput("password", "Password:")
```

plotOutput — *Create an plot output element*

Description

Render a renderPlot within an application page.

Usage

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

Arguments

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

Value

A plot output element that can be included in a panel

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.

Examples

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

---

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

Description

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

Usage

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

Arguments

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

Details

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

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

`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 tham `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`. `NULL` hides the progress bar, if it is currently visible.
`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.

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

```
## Not run:
# server.R
shinyServer(function(input, output, session) {
  output$plot <- renderPlot({
    progress <- shiny::Progress$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)
  })
})

## End(Not run)
```

`radioButtons` *Create radio buttons*

Description

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

Usage

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

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)
`selected`	The initially selected value (if not specified then defaults to the first value)
`inline`	If `TRUE`, render the choices inline (i.e. horizontally)

Value

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

See Also

updateRadioButtons

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

Examples

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

---

reactive *Create a reactive expression*

Description

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

Usage

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

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.

Details

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

See the Shiny tutorial for more information about reactive expressions.

Value

a function, wrapped in a S3 class "reactive"

Examples

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

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

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

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

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

`reactiveFileReader` *Reactive file reader*

Description

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

Usage

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

Arguments

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

Details

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

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

Value

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

See Also

reactivePoll

Examples

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

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

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

## End(Not run)
```

---

reactivePlot *Plot output (deprecated)*

Description

See `renderPlot`.

Usage

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

Arguments

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

---

reactivePoll *Reactive polling*

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

Value

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

See Also

`reactiveFileReader`

Examples

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

## End(Not run)
```

`reactivePrint` *Print output (deprecated)*

Description

See `renderPrint`.

Usage

```
reactivePrint(func)
```

Arguments

`func`	A function.

`reactiveTable` *Table output (deprecated)*

Description

See `renderTable`.

Usage

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

Arguments

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

`reactiveText` *Text output (deprecated)*

Description

See `renderText`.

Usage

```
reactiveText(func)
```

Arguments

`func`	A function.

reactiveTimer *Timer*

Description

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

Usage

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

Arguments

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

Details

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

See `invalidateLater` as a safer and simpler alternative.

Value

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

See Also

`invalidateLater`

Examples

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

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

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

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

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

## End(Not run)
```

`reactiveUI` *UI output (deprecated)*

Description

See `renderUI`.

Usage

```
reactiveUI(func)
```

Arguments

`func`	A function.

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

Description

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

---

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 RJSONIO) 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 RJSONIO. 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 RJSONIO 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`

`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)
```

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.

Details

For the `options` argument, the character elements that have the class `"AsIs"` (usually returned from `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. We may deprecate `renderDataTable()` and `dataTableOutput()` in the future when the **DT** package is mature enough.

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)
```

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`.

Details

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

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

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

See Also

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

Examples

```
## Not run:

shinyServer(function(input, output, clientData) {

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

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

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

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

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

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

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

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

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


## End(Not run)
```

`renderPlot` *Plot Output*

Description

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

Usage

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

Arguments

`expr`	An expression that generates a plot.

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

Details

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

See Also

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

`renderPrint` *Printable Output*

Description

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

Usage

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

Arguments

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

Details

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

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

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

See Also

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

Examples

```
isolate({

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

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

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

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

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

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


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

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

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

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

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

})
```

`renderTable` *Table Output*

Description

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

Usage

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

Arguments

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

Details

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

renderText *Text Output*

Description

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

Usage

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

Arguments

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

Details

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

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

See Also

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

Examples

```
isolate({

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

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

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

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

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

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


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

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

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

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

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

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

})
```

`renderUI` *UI Output*

Description

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

Usage

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

Arguments

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

Details

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

See Also

conditionalPanel

Examples

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

    )
  })

## End(Not run)
```

`repeatable` *Make a random number generator repeatable*

Description

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

Usage

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

Arguments

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

Value

A repeatable version of the function that was passed in.

Note

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

Examples

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

`runApp` *Run Shiny Application*

Description

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

Usage

```
runApp(appDir = getwd(), port = NULL,
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"), workerId = "",
  quiet = FALSE, display.mode = c("auto", "normal", "showcase"))
```

Arguments

`appDir`	The directory of the application. Should contain `server.R`, plus, either `ui.R` or a `www` directory that contains the file `index.html`. Defaults to the working directory. Instead of a directory, this could be a list with `ui` and `server` components, or a Shiny app object created by `shinyApp`.
`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. 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.

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()) {
  # 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")
}
```

---

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, ...)

runGist(gist, ...)

runGitHub(repo, username = getOption("github.user"), ref = "master",
  subdir = 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"`.
`...`	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/")
}
```

`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.
`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 `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/brianreavis/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 select list control that can be added to a UI definition.

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`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `submitButton`; `textInput`

Examples

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

`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

`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`, and `url_hash_initial` 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.

`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 implements 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.

`request` An environment that implements the Rook specification for HTTP requests. This is the request that was used to initiate the websocket connection (as opposed to the request that downloaded the web page for the app).

`sendCustomMessage(type, message)`
Sends a custom message to the web page. `type` must be a single-element character vector giving the type of message, while `message` can be any RJSONIO-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.

`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`.

`shiny-options` *Global options for Shiny*

Description

There are a number of global options that affect Shiny's behavior. These can be set with (for example) `options(shiny.trace=TRUE)`.

Details

- **shiny.launch.browser** A boolean which controls the default behavior when an app is run. See `runApp` for more information.
- **shiny.trace** If `TRUE`, all of the messages sent between the R server and the web browser client will be printed on the console. This is useful for debugging.
- **shiny.reactlog** If `TRUE`, enable logging of reactive events, which can be viewed later with the `showReactLog` function. This incurs a substantial performance penalty and should not be used in production.
- **shiny.usecairo** This is used to disable graphical rendering by the Cairo package, if it is installed. See `plotPNG` for more information.
- **shiny.maxRequestSize** This is a number which specifies the maximum web request size, which serves as a size limit for file uploads. If unset, the maximum request size defaults to 5MB.
- **shiny.suppressMissingContextError** 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.host** The IP address that Shiny should listen on. See `runApp` for more information.
- **shiny.json.digits** The number of digits to use when converting numbers to JSON format to send to the client web browser.
- **shiny.error** 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.observer.error** This can be a function that is called by an observer when an unhandled error occurs in it or an upstream reactive. By default, these errors will result in a warning at the console, and the websocket connection will close.
- **shiny.table.class** CSS class names to use for tables.
- **shiny.deprecation.messages** This controls whether messages for deprecated functions in Shiny will be printed. See `shinyDeprecated` for more information.

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`). You generally shouldn't need to use these functions to create/run applications; they are intended for interoperability purposes, such as embedding Shiny apps inside a **knitr** document.

Usage

```
shinyApp(ui = NULL, server = NULL, onStart = NULL, options = list(),
  uiPattern = "/")

shinyAppDir(appDir, options = list())

as.shiny.appobj(x)

## S3 method for class 'shiny.appobj'
as.shiny.appobj(x)

## S3 method for class 'list'
as.shiny.appobj(x)

## S3 method for class 'character'
as.shiny.appobj(x)

is.shiny.appobj(x)

## S3 method for class 'shiny.appobj'
print(x, ...)

## S3 method for class 'shiny.appobj'
as.tags(x, ...)
```

Arguments

`ui`	The UI definition of the app (for example, a call to `fluidPage()` with nested controls)
`server`	A server function
`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. You can also specify `width` and `height` parameters which provide 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 successful.
`appDir`	Path to directory that contains a Shiny app (i.e. a server.R file and either ui.R or www/index.html)
`x`	Object to convert to a Shiny app.
`...`	Additional parameters to be passed to print.

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()) {
  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)
}
```

shinyServer *Define Server Functionality*

Description

Defines the server-side logic of the Shiny application. This generally involves creating functions that map user inputs to various kinds of output. In older versions of Shiny, it was necessary to call `shinyServer()` in the `server.R` file, but this is no longer required as of Shiny 0.10. Now the `server.R` file may simply return the appropriate server function (as the last expression in the code), without calling `shinyServer()`.

Usage

```
shinyServer(func)
```

Arguments

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

Details

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

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

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

Examples

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


# It is also possible for a server.R file to simply return the function,
# without calling shinyServer().
# For example, the server.R file could contain just the following:
function(input, output, session) {
  output$uppercase <- renderText({
    toupper(input$message)
  })
}

## End(Not run)
```

---

shinyUI *Create a Shiny UI handler*

Description

Historically this function was used in ui.R files to register a user interface with Shiny. It is no longer required as of Shiny 0.10; simply ensure that the last expression to be returned from ui.R is a user interface. This function is kept for backwards compatibility with older applications. It returns the value that is passed to it.

Usage

```
shinyUI(ui)
```

Arguments

`ui`	A user interace definition

Value

The user interface definition, without modifications or side effects.

---

showReactLog *Reactive Log Visualizer*

Description

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

Usage

```
showReactLog()
```

Details

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

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

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

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

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

`sidebarLayout` *Layout a sidebar and main area*

Description

Create a layout with a sidebar and main area. 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)
```

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.

Examples

```
# Define UI
shinyUI(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")
    )
  )
))
```

`sidebarPanel` *Create a sidebar panel*

Description

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

Usage

```
sidebarPanel(..., width = 4)
```

Arguments

`...`	UI elements to include on the sidebar
`width`	The width of the sidebar. For fluid layouts this is out of 12 total units; for fixed layouts it is out of whatever the width of the sidebar's parent column is.

Value

A sidebar that can be passed to `sidebarLayout`

Examples

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

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

`singleton` *Include content only once*

Description

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

Usage

```
singleton(x, value = TRUE)

is.singleton(x)
```

Arguments

`x`	A `tag`, text, `HTML`, or list.
`value`	Whether the object should be a singleton.

`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)

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).
`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.
`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.

See Also

`updateSliderInput`

Other input.elements: `actionButton`, `actionLink`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`; `textInput`

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.

Examples

```
# Equal sizing
splitLayout(
  plotOutput("plot1"),
  plotOutput("plot2")
)

# Custom widths
splitLayout(cellWidths = c("25%", "75%"),
  plotOutput("plot1"),
  plotOutput("plot2")
)

# All cells at 300 pixels wide, with cell padding
# and a border around everything
splitLayout(
  style = "border: 1px solid silver;",
  cellWidths = 300,
  cellArgs = list(style = "padding: 6px"),
  plotOutput("plot1"),
  plotOutput("plot2"),
  plotOutput("plot3")
)
```

stopApp — *Stop the currently running Shiny app*

Description

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

Usage

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

Arguments

`returnValue`	The value that should be returned from `runApp`.

submitButton — *Create a submit button*

Description

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

Usage

```
submitButton(text = "Apply Changes", icon = NULL)
```

Arguments

`text`	Button caption
`icon`	Optional `icon` to appear on the button

Value

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

See Also

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `textInput`

Examples

```
submitButton("Update View")
submitButton("Update View", icon("refresh"))
```

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`.

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 = c("above", "below", "left", "right"))
```

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	The position of the tabs relative to the content. Valid values are "above", "below", "left", and "right" (defaults to "above"). Note that the position argument is not valid when type is "pill".

Value

A tabset that can be passed to mainPanel

See Also

tabPanel, updateTabsetPanel

Examples

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

`tag` *HTML Tag Object*

Description

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

Usage

```
tagList(...)

tagAppendAttributes(tag, ...)

tagAppendChild(tag, child)

tagAppendChildren(tag, ..., list = NULL)

tagSetChildren(tag, ..., list = NULL)

tag(`_tag_name`, varArgs)
```

Arguments

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

Value

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

Examples

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

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

`textInput` *Create a text input control*

Description

Create an input control for entry of unstructured text values

Usage

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

Arguments

`inputId`	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.

Value

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

See Also

`updateTextInput`

Other input.elements: `actionButton`, `actionLink`; `animationOptions`, `sliderInput`; `checkboxGroupInput`; `checkboxInput`; `dateInput`; `dateRangeInput`; `fileInput`; `numericInput`; `passwordInput`; `radioButtons`; `selectInput`, `selectizeInput`; `submitButton`

Examples

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

`textOutput` *Create a text output element*

Description

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

```
titlePanel("Hello Shiny!")
```

---

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)
```

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.
`selected`	The values that should be initially selected, if any.
`inline`	If `TRUE`, render the choices inline (i.e. horizontally)

Details

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

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

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

See Also

`checkboxGroupInput`

Examples

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

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

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

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

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

## End(Not run)
```

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

Description

Change the value of a checkbox input on the client

Usage

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

Arguments

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

Details

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

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

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

See Also

`checkboxInput`

Examples

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

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

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

## End(Not run)
```

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

Description

Change the value of a date input on the client

Usage

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

Arguments

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

`min`	The minimum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.
`max`	The maximum allowed date. Either a Date object, or a string in `yyyy-mm-dd` format.

Details

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

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

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

See Also

`dateInput`

Examples

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

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

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

## End(Not run)
```

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

Description

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

Usage

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

Arguments

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

Details

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

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

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

See Also

`dateRangeInput`

Examples

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

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

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

## End(Not run)
```

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

Description

Change the value of a number input on the client

Usage

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

Arguments

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

Details

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

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

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

See Also

`numericInput`

Examples

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

  observe({
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller
    updateNumericInput(session, "inNumber", value = x)

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

## End(Not run)
```

---

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

Description

Change the value of a radio input on the client

Usage

```
updateRadioButtons(session, inputId, label = NULL, choices = NULL,
  selected = NULL, inline = 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)
selected	The initially selected value (if not specified then defaults to the first value)
inline	If TRUE, render the choices inline (i.e. horizontally)

Details

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

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

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

See Also

radioButtons

Examples

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

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

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

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

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

## End(Not run)
```

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

Description

Change the value of a select input on the client

Usage

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

updateSelectizeInput(session, inputId, label = NULL, choices = 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.
`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 `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**)

Details

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

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

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

See Also

`selectInput`

Examples

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

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

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

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

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

## End(Not run)
```

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

Description

Change the value of a slider input on the client

Usage

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

Arguments

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

Details

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

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

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

See Also

`sliderInput`

Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      sidebarLayout(
        sidebarPanel(
          p("The first slider controls the second"),
          slider2Input("control", "Controller:", min=0, max=20, value=10,
                       step=1),
          slider2Input("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)
```

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

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

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

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

## End(Not run)
```

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

Description

Change the value of a text input on the client

Usage

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

Arguments

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

Details

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

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

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

See Also

textInput

Examples

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

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

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

    # Can also set the label, this time for input$inText2
    updateTextInput(session, "inText2",
      label = paste("New label", x),
      value = paste("New text", x))
  })
})

## End(Not run)
```

---

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

```
# in ui.R
fluidPage(
  checkboxGroupInput('in1', 'Check some letters', choices = head(LETTERS)),
  selectizeInput('in2', 'Select a state', choices = state.name),
  plotOutput('plot')
)
```

```
# in server.R
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 = ', '))
  })
}
```

`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 specified.

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"`, or a number. If the number has a suffix, it must be valid: `px`, `%`, `em`, `pt`, `in`, `cm`, `mm`, `ex`, or `pc`. 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'
```

`verbatimTextOutput` *Create a verbatim text output element*

Description

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

Usage

```
verbatimTextOutput(outputId)
```

Arguments

`outputId`	output variable to read the value from

Details

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

Value

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

Examples

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

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

`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

`fluidPage`, `flowLayout`

Examples

```
shinyUI(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")
  )
))
```

`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, 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`. `NULL` hides the progress bar, if it is currently visible.
`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`.
`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.

See Also

`Progress`

Examples

```
## Not run:
# server.R
shinyServer(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)
  })
})

## End(Not run)
```

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

∗Topic **datasets**
    Progress, 54
.Random.seed, *77*

a *(*builder*)*, 9
absolutePanel, 5
actionButton, 7, *11*, *12*, *16*, *18*, *24*, *43*, *47*, *51*, *56*, *82*, *93*, *95*, *100*, *115*
actionLink, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*
actionLink *(*actionButton*)*, 7
addResourcePath, 8, *15*
animationOptions, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*
animationOptions *(*sliderInput*)*, 92
as.character, *10*, *99*
as.list, *64*
as.shiny.appobj *(*shinyApp*)*, 86
as.tags.shiny.appobj *(*shinyApp*)*, 86

basicPage *(*bootstrapPage*)*, 8
bootstrapPage, 8
br *(*builder*)*, 9
builder, 9

CairoPNG, *53*
capture.output, *72*, *74*
cat, *74*
checkboxGroupInput, *7*, 11, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*, *102*
checkboxInput, *7*, *11*, 12, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*, *104*
code *(*builder*)*, 9
column, 12, *24–26*
conditionalPanel, 13
createWebDependency, 15

dataTableOutput *(*tableOutput*)*, 96
dateInput, *7*, *11*, *12*, 15, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*, *105*
dateRangeInput, *7*, *11*, *12*, *16*, 17, *24*, *43*, *51*, *56*, *82*, *93*, *95*, *100*, *106*
div *(*builder*)*, 9
domain, *44*
domains, 19, *44*, *46*, *56*
downloadButton, 20, *21*
downloadHandler, *20*, 21
downloadLink, *21*
downloadLink *(*downloadButton*)*, 20

em *(*builder*)*, 9
eventReactive, *7*
eventReactive *(*observeEvent*)*, 45
exprToFunction, 22, *34*

fileInput, *7*, *11*, *12*, *16*, *18*, 23, *43*, *51*, *56*, *82*, *93*, *95*, *100*
fixedPage, *9*, 24
fixedPanel *(*absolutePanel*)*, 5
fixedRow, *12*, *13*
fixedRow *(*fixedPage*)*, 24
flowLayout, 25, *33*, *118*
fluidPage, *9*, *24*, 26, *49*, *118*
fluidRow, *9*, *12*, *13*
fluidRow *(*fluidPage*)*, 26

getDefaultReactiveDomain *(*domains*)*, 19
graphics, *52*
grid, *52*

h1 *(*builder*)*, 9
h2 *(*builder*)*, 9
h3 *(*builder*)*, 9
h4 *(*builder*)*, 9
h5 *(*builder*)*, 9
h6 *(*builder*)*, 9
headerPanel, 27, *49*
helpText, 28
hr *(*builder*)*, 9
HTML, *10*, 29, *76*, *92*, *93*, *99*

htmlDependency, *15*
htmlOutput, 29

I, *67*, *82*, *110*
icon, *7*, 30, *95*
imageOutput, 31
img *(*builder*)*, 9
include, 32
includeCSS *(*include*)*, 32
includeHTML *(*include*)*, 32
includeMarkdown *(*include*)*, 32
includeScript *(*include*)*, 32
includeText *(*include*)*, 32
incProgress *(*withProgress*)*, 119
inputPanel, 33
installExprFunction, 33
invalidateLater, 34, *62*
invisible, *71*, *72*
is.reactive *(*reactive*)*, 56
is.reactivevalues, 35, *64*
is.shiny.appobj *(*shinyApp*)*, 86
is.singleton *(*singleton*)*, 92
isolate, 35, *40*, *46*, *47*, *64*, *85*

knit_print.html, 37
knit_print.shiny.appobj *(*knitr_methods*)*, 37
knit_print.shiny.render.function *(*knitr_methods*)*, 37
knit_print.shiny.tag *(*knit_print.html*)*, 37
knitr_methods, 37

local, *36*

mainPanel, 38, *49*, *90*, *98*
makeReactiveBinding, 38
markRenderFunction, 39
maskReactiveContext, 40

navbarMenu *(*navbarPage*)*, 40
navbarPage, *30*, 40, *97*, *112*
navlistPanel, 42, *112*
need *(*validate*)*, 114
numericInput, *7*, *11*, *12*, *16*, *18*, *24*, 43, *51*, *56*, *82*, *93*, *95*, *100*, *102*, *104–108*, *110*, *111*, *113*

observe, *19*, 44, *47*
observeEvent, *7*, 45

onReactiveDomainEnded *(*domains*)*, 19
options, *71*
outputOptions, 48

p *(*builder*)*, 9
pageWithSidebar, *28*, 49
parseQueryString, 50, *84*
passwordInput, *7*, *11*, *12*, *16*, *18*, *24*, *43*, 51, *56*, *82*, *93*, *95*, *100*
plotOutput, 52
plotPNG, 53, *69*, *71*, *85*
png, *53*, *71*
pre, *32*
pre *(*builder*)*, 9
print.shiny.appobj *(*shinyApp*)*, 86
print.xtable, *73*
Progress, 54, *120*

radioButtons, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, 55, *82*, *93*, *95*, *100*, *108*
reactive, *19*, *47*, 56
reactive expression, *47*
Reactive expressions, *62*
reactiveFileReader, 57, *60*
reactivePlot, 59
reactivePoll, *58*, 59
reactivePrint, 60
reactiveTable, 61
reactiveText, 61
reactiveTimer, *34*, 62
reactiveUI, 63
reactiveValues, *35*, 63, *83*
reactiveValuesToList, 64
registerInputHandler, 65, *67*
removeInputHandler, *66*, 66
renderDataTable, 67, *82*, *96*, *110*
renderImage, *31*, 68
renderPlot, *52*, *59*, 70, *114*
renderPrint, *60*, 71, *74*, *117*
renderTable, *61*, 73, *96*
renderText, *61*, *72*, 74, *101*
renderUI, *63*, 75, *119*
repeatable, 76
runApp, 77, *80*, *85*, *87*, *90*, *95*
runExample, 79
runGist *(*runUrl*)*, 80
runGitHub *(*runUrl*)*, 80
runUrl, 80

selectInput, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, 81, *93*, *95*, *100*, *110*
selectizeInput, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *93*, *95*, *100*
selectizeInput (selectInput), 81
serverInfo, 83
session, 83
setProgress, *54*
setProgress (withProgress), 119
shiny (shiny-package), 5
shiny-options, *5*, 85
shiny-package, 5
shinyApp, *78*, 86
shinyAppDir (shinyApp), 86
shinyDeprecated, *85*
shinyServer, 88
shinyUI, *9*, *24*, *26*, *41*, *49*, 89
showReactLog, *85*, 89
sidebarLayout, *9*, *26*, *38*, *49*, 90, *91*
sidebarPanel, *49*, *90*, 91
singleton, *8*, 92
sliderInput, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, 92, *95*, *100*, *111*
span (builder), 9
splitLayout, 94
stopApp, 95
strong (builder), 9
submitButton, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, 95, *100*
Sys.time, *62*

tableOutput, 96
tabPanel, *30*, *40–42*, 97, *98*
tabsetPanel, *41*, *97*, 98, *112*
tag, *10*, *29*, *92*, *93*, 99
tagAppendAttributes (tag), 99
tagAppendChild (tag), 99
tagAppendChildren (tag), 99
tagList (tag), 99
tags, *99*
tags (builder), 9
tagSetChildren (tag), 99
textInput, *7*, *11*, *12*, *16*, *18*, *24*, *43*, *51*, *56*, *82*, *93*, *95*, 100, *114*
textOutput, 101
titlePanel, *26*, 101

uiOutput, *76*
uiOutput (htmlOutput), 29
updateCheckboxGroupInput, *11*, 102
updateCheckboxInput, *12*, 103
updateDateInput, *16*, 104
updateDateRangeInput, *18*, 105
updateNumericInput, *43*, 107
updateRadioButtons, *56*, 108
updateSelectInput, *82*, 109
updateSelectizeInput (updateSelectInput), 109
updateSliderInput, *93*, 111
updateTabsetPanel, *98*, 112
updateTextInput, *51*, *85*, *100*, 113

validate, 114
validateCssUnit, *82*, *93*, *94*, 116
validation, *47*
verbatimTextOutput, 117
verticalLayout, *25*, 117

wellPanel, 118
withMathJax, 119
withProgress, *54*, *55*, 119
withReactiveDomain (domains), 19
withTags, 121

xtable, *73*