Package ‘shiny’

August 1, 2024

Type Package

Title Web Application Framework for R

Version 1.9.1

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

License GPL-3 | file LICENSE

Depends R (>= 3.0.2), methods

Imports utils, grDevices, httpuv (>= 1.5.2), mime (>= 0.3), jsonlite (>= 0.9.16), xtable, fontawesome (>= 0.4.0), htmltools (>= 0.5.4), R6 (>= 2.0), soucetools, later (>= 1.0.0), promises (>= 1.1.0), tools, crayon, rlang (>= 0.4.10), fastmap (>= 1.1.1), withr, commonmark (>= 1.7), glue (>= 1.3.2), bslib (>= 0.6.0), cachem (>= 1.1.0), lifecycle (>= 0.2.0)

Suggests datasets, DT, Cairo (>= 1.5-5), testthat (>= 3.0.0), knitr (>= 1.6), markdown, rmarkdown, ggplot2, reactlog (>= 1.0.0), magrittr, yaml, future, dygraphs, ragg, showtext, sass


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

'input-submit.R' 'input-text.R' 'input-textarea.R'
'input-utils.R' 'insert-tab.R' 'insert-ui.R' 'jqueryui.R'
'knitr.R' 'middleware-shiny.R' 'middleware.R' 'timer.R'
'shiny.R' 'mock-session.R' 'modal.R' 'modules.R'
'notifications.R' 'priorityqueue.R' 'progress.R' 'react.R'
'reexports.R' 'render-cached-plot.R' 'render-plot.R'
'render-table.R' 'run-url.R' 'runapp.R' 'serializers.R'
'server-input-handlers.R' 'server-resource-paths.R' 'server.R'
'shiny-options.R' 'shiny-package.R' 'shinyapp.R' 'shinyui.R'
'shinywrappers.R' 'showcase.R' 'snapshot.R' 'staticimports.R'
'tar.R' 'test-export.R' 'test-server.R' 'test.R'
'update-input.R' 'utils-lang.R' 'version_bs_date_picker.R'
'version_ion_range_slider.R' 'version_jquery.R'
'version_jqueryui.R' 'version_selectize.R' 'version_strftime.R'
'viewer.R'

RoxygenNote 7.3.2
Encoding UTF-8
RdMacros lifecycle

Config/testthat/edition 3
Config/Needs/check shinytest2

NeedsCompilation no

Author Winston Chang [aut, cre] (<https://orcid.org/0000-0002-1576-2126>),
  Joe Cheng [aut],
  JJ Allaire [aut],
  Carson Sievert [aut] (<https://orcid.org/0000-0002-4958-2844>),
  Barret Schloerke [aut] (<https://orcid.org/0000-0001-9986-114X>),
  Yihui Xie [aut],
  Jeff Allen [aut],
  Jonathan McPherson [aut],
  Alan Dipert [aut],
  Barbara Borges [aut],
  Posit Software, PBC [cph, fnd],
  jQuery Foundation [cph] (jQuery library and jQuery UI library),
  jQuery contributors [ctb, cph] (jQuery library; authors listed in
  inst/www/shared/jquery-AUTHORS.txt),
  jQuery UI contributors [ctb, cph] (jQuery UI library; authors listed in
  inst/www/shared/jqueryui/AUTHORS.txt),
  Mark Otto [ctb] (Bootstrap library),
  Jacob Thornton [ctb] (Bootstrap library),
  Bootstrap contributors [ctb] (Bootstrap library),
  Twitter, Inc [cph] (Bootstrap library),
  Prem Nawaz Khan [ctb] (Bootstrap accessibility plugin),
  Victor Tsaran [ctb] (Bootstrap accessibility plugin),
  Dennis Lembree [ctb] (Bootstrap accessibility plugin),
  Srinivas Chakravarthula [ctb] (Bootstrap accessibility plugin),
  Cathy O'Connor [ctb] (Bootstrap accessibility plugin),
Contents

PayPal, Inc [cph] (Bootstrap accessibility plugin),
Stefan Petre [ctb, cph] (Bootstrap-datepicker library),
Andrew Rowls [ctb, cph] (Bootstrap-datepicker library),
Brian Reavis [ctb, cph] (selectize.js library),
Salmen Bejaoui [ctb, cph] (selectize-plugin-a11y library),
Denis Ineshin [ctb, cph] (ion.rangeSlider library),
Sami Samhuri [ctb, cph] (Javascript strftime library),
SpryMedia Limited [ctb, cph] (DataTables library),
John Fraser [ctb, cph] (showdown.js library),
John Gruber [ctb, cph] (showdown.js library),
Ivan Sagalaev [ctb, cph] (highlight.js library),
R Core Team [ctb, cph] (tar implementation from R)

Maintainer Winston Chang <winston@posit.co>
Repository CRAN
Date/Publication 2024-08-01 10:50:02 UTC

Contents

shiny-package .................................................. 6
absolutePanel .................................................... 8
actionButton ..................................................... 9
addResourcePath ................................................ 11
bindCache ......................................................... 12
bindEvent ......................................................... 19
bookmarkButton ................................................ 22
bootstrapLib ...................................................... 23
bootstrapPage ................................................... 24
brushedPoints ................................................... 25
brushOpts ......................................................... 27
busyIndicatorOptions ......................................... 28
callModule ......................................................... 31
checkboxGroupInput ........................................... 32
checkboxInput .................................................... 34
clickOpts ......................................................... 35
column .......................................................... 36
conditionalPanel ............................................... 37
createRenderFunction ........................................ 38
createWebDependency ......................................... 42
dateInput ......................................................... 43
dateRangeInput ................................................ 45
debounce ......................................................... 48
devmode ........................................................ 50
domains .......................................................... 54
downloadButton ................................................ 55
downloadHandler ............................................... 57
enableBookmarking ........................................... 58
exportTestValues .............................................. 62
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtendedTask</td>
<td>63</td>
</tr>
<tr>
<td>fileInput</td>
<td>66</td>
</tr>
<tr>
<td>fillPage</td>
<td>68</td>
</tr>
<tr>
<td>fillRow</td>
<td>70</td>
</tr>
<tr>
<td>fixedPage</td>
<td>71</td>
</tr>
<tr>
<td>flowLayout</td>
<td>73</td>
</tr>
<tr>
<td>fluidPage</td>
<td>73</td>
</tr>
<tr>
<td>freezeReactiveVal</td>
<td>75</td>
</tr>
<tr>
<td>getCurrentOutputInfo</td>
<td>77</td>
</tr>
<tr>
<td>getQueryString</td>
<td>78</td>
</tr>
<tr>
<td>getShinyOption</td>
<td>80</td>
</tr>
<tr>
<td>helpText</td>
<td>83</td>
</tr>
<tr>
<td>htmlOutput</td>
<td>84</td>
</tr>
<tr>
<td>icon</td>
<td>85</td>
</tr>
<tr>
<td>inputPanel</td>
<td>86</td>
</tr>
<tr>
<td>insertTab</td>
<td>86</td>
</tr>
<tr>
<td>insertUI</td>
<td>89</td>
</tr>
<tr>
<td>invalidateLater</td>
<td>91</td>
</tr>
<tr>
<td>is.reactivevalues</td>
<td>93</td>
</tr>
<tr>
<td>isolate</td>
<td>93</td>
</tr>
<tr>
<td>isRunning</td>
<td>94</td>
</tr>
<tr>
<td>isTruthy</td>
<td>95</td>
</tr>
<tr>
<td>loadSupport</td>
<td>96</td>
</tr>
<tr>
<td>markdown</td>
<td>96</td>
</tr>
<tr>
<td>markRenderFunction</td>
<td>98</td>
</tr>
<tr>
<td>maskReactiveContext</td>
<td>99</td>
</tr>
<tr>
<td>MockShinySession</td>
<td>100</td>
</tr>
<tr>
<td>modalDialog</td>
<td>106</td>
</tr>
<tr>
<td>moduleServer</td>
<td>109</td>
</tr>
<tr>
<td>navbarPage</td>
<td>111</td>
</tr>
<tr>
<td>navlistPanel</td>
<td>113</td>
</tr>
<tr>
<td>NS</td>
<td>115</td>
</tr>
<tr>
<td>numericInput</td>
<td>116</td>
</tr>
<tr>
<td>observe</td>
<td>117</td>
</tr>
<tr>
<td>observeEvent</td>
<td>119</td>
</tr>
<tr>
<td>onBookmark</td>
<td>124</td>
</tr>
<tr>
<td>onFlush</td>
<td>128</td>
</tr>
<tr>
<td>onStop</td>
<td>130</td>
</tr>
<tr>
<td>outputOptions</td>
<td>131</td>
</tr>
<tr>
<td>parseQueryString</td>
<td>132</td>
</tr>
<tr>
<td>passwordInput</td>
<td>133</td>
</tr>
<tr>
<td>plotOutput</td>
<td>134</td>
</tr>
<tr>
<td>plotPNG</td>
<td>139</td>
</tr>
<tr>
<td>Progress</td>
<td>140</td>
</tr>
<tr>
<td>radioButtons</td>
<td>143</td>
</tr>
<tr>
<td>reactive</td>
<td>146</td>
</tr>
<tr>
<td>reactiveFileReader</td>
<td>147</td>
</tr>
<tr>
<td>reactivePoll</td>
<td>149</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
</tr>
<tr>
<td>reactiveTimer</td>
<td>150</td>
</tr>
<tr>
<td>reactiveVal</td>
<td>152</td>
</tr>
<tr>
<td>reactiveValues</td>
<td>153</td>
</tr>
<tr>
<td>reactiveValuesToList</td>
<td>154</td>
</tr>
<tr>
<td>reactLog</td>
<td>155</td>
</tr>
<tr>
<td>registerInputHandler</td>
<td>157</td>
</tr>
<tr>
<td>removeInputHandler</td>
<td>158</td>
</tr>
<tr>
<td>renderCachedPlot</td>
<td>159</td>
</tr>
<tr>
<td>renderImage</td>
<td>163</td>
</tr>
<tr>
<td>renderPlot</td>
<td>165</td>
</tr>
<tr>
<td>renderPrint</td>
<td>167</td>
</tr>
<tr>
<td>renderUI</td>
<td>169</td>
</tr>
<tr>
<td>repeatable</td>
<td>171</td>
</tr>
<tr>
<td>req</td>
<td>171</td>
</tr>
<tr>
<td>restoreInput</td>
<td>173</td>
</tr>
<tr>
<td>runApp</td>
<td>174</td>
</tr>
<tr>
<td>runExample</td>
<td>176</td>
</tr>
<tr>
<td>runGadget</td>
<td>177</td>
</tr>
<tr>
<td>runTests</td>
<td>178</td>
</tr>
<tr>
<td>runUrl</td>
<td>179</td>
</tr>
<tr>
<td>safeError</td>
<td>181</td>
</tr>
<tr>
<td>selectInput</td>
<td>182</td>
</tr>
<tr>
<td>serverInfo</td>
<td>185</td>
</tr>
<tr>
<td>session</td>
<td>185</td>
</tr>
<tr>
<td>setBookmarkExclude</td>
<td>188</td>
</tr>
<tr>
<td>setSerializer</td>
<td>189</td>
</tr>
<tr>
<td>shinyApp</td>
<td>189</td>
</tr>
<tr>
<td>shinyAppTemplate</td>
<td>191</td>
</tr>
<tr>
<td>showBookmarkUrlModal</td>
<td>193</td>
</tr>
<tr>
<td>showModal</td>
<td>193</td>
</tr>
<tr>
<td>showNotification</td>
<td>194</td>
</tr>
<tr>
<td>showTab</td>
<td>195</td>
</tr>
<tr>
<td>sidebarLayout</td>
<td>197</td>
</tr>
<tr>
<td>sizeGrowthRatio</td>
<td>199</td>
</tr>
<tr>
<td>sliderInput</td>
<td>200</td>
</tr>
<tr>
<td>snapshotExclude</td>
<td>202</td>
</tr>
<tr>
<td>snapshotPreprocessInput</td>
<td>202</td>
</tr>
<tr>
<td>snapshotPreprocessOutput</td>
<td>203</td>
</tr>
<tr>
<td>splitLayout</td>
<td>203</td>
</tr>
<tr>
<td>stopApp</td>
<td>205</td>
</tr>
<tr>
<td>submitButton</td>
<td>205</td>
</tr>
<tr>
<td>tableOutput</td>
<td>206</td>
</tr>
<tr>
<td>tabPanel</td>
<td>208</td>
</tr>
<tr>
<td>tabsetPanel</td>
<td>209</td>
</tr>
<tr>
<td>testServer</td>
<td>211</td>
</tr>
<tr>
<td>textAreaInput</td>
<td>212</td>
</tr>
<tr>
<td>textInput</td>
<td>214</td>
</tr>
<tr>
<td>textOutput</td>
<td>215</td>
</tr>
</tbody>
</table>
Description

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

Details

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

Author(s)

Maintainer: Winston Chang <winston@posit.co> (ORCID)

Authors:

- Joe Cheng <joe@posit.co>
- JJ Allaire <jj@posit.co>
shiny-package

- Carson Sievert <carson@posit.co> (ORCID)
- Barret Schloerke <barret@posit.co> (ORCID)
- Yihui Xie <yihui@posit.co>
- Jeff Allen
- Jonathan McPherson <jonathan@posit.co>
- Alan Dipert
- Barbara Borges

Other contributors:

- Posit Software, PBC [copyright holder, funder]
- jQuery Foundation (jQuery library and jQuery UI library) [copyright holder]
- jQuery contributors (jQuery library; authors listed in inst/www/shared/jquery-AUTHORS.txt) [contributor, copyright holder]
- jQuery UI contributors (jQuery UI library; authors listed in inst/www/shared/jqueryui/AUTHORS.txt) [contributor, copyright holder]
- Mark Otto (Bootstrap library) [contributor]
- Jacob Thornton (Bootstrap library) [contributor]
- Bootstrap contributors (Bootstrap library) [contributor]
- Twitter, Inc (Bootstrap library) [copyright holder]
- Prem Nawaz Khan (Bootstrap accessibility plugin) [contributor]
- Victor Tsaran (Bootstrap accessibility plugin) [contributor]
- Dennis Lembree (Bootstrap accessibility plugin) [contributor]
- Srinivasu Chakravarthula (Bootstrap accessibility plugin) [contributor]
- Cathy O’Connor (Bootstrap accessibility plugin) [contributor]
- PayPal, Inc (Bootstrap accessibility plugin) [copyright holder]
- Stefan Petre (Bootstrap-datepicker library) [contributor, copyright holder]
- Andrew Rowls (Bootstrap-datepicker library) [contributor, copyright holder]
- Brian Reavis (selectize.js library) [contributor, copyright holder]
- Salmen Bejaoui (selectize-plugin-a11y library) [contributor, copyright holder]
- Denis Ineshin (ion.rangeSlider library) [contributor, copyright holder]
- Sami Samhuri (Javascript strftime library) [contributor, copyright holder]
- SpryMedia Limited (DataTables library) [contributor, copyright holder]
- John Fraser (showdown.js library) [contributor, copyright holder]
- John Gruber (showdown.js library) [contributor, copyright holder]
- Ivan Sagalaev (highlight.js library) [contributor, copyright holder]
- R Core Team (tar implementation from R) [contributor, copyright holder]

See Also

shiny-options for documentation about global options.
absolutePanel  
*Panel with absolute positioning*

**Description**

Creates a panel whose contents are absolutely positioned.

**Usage**

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

```r
fixedPanel(
  ..., 
  top = NULL, 
  left = NULL, 
  right = NULL, 
  bottom = NULL, 
  width = NULL, 
  height = NULL, 
  draggable = FALSE, 
  fixed = 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.
**ActionButton**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>Width of the panel.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the panel.</td>
</tr>
<tr>
<td>draggable</td>
<td>If TRUE, allows the user to move the panel by clicking and dragging.</td>
</tr>
<tr>
<td>fixed</td>
<td>Positions the panel relative to the browser window and prevents it from being scrolled with the rest of the page.</td>
</tr>
<tr>
<td>cursor</td>
<td>The type of cursor that should appear when the user mouses over the panel. Use &quot;move&quot; for a north-east-south-west icon, &quot;default&quot; for the usual cursor arrow, or &quot;inherit&quot; for the usual cursor behavior (including changing to an I-beam when the cursor is over text). The default is &quot;auto&quot;, which is equivalent to ifelse(draggable, &quot;move&quot;, &quot;inherit&quot;).</td>
</tr>
</tbody>
</table>

**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**

```r
actionButton(inputId, label, icon = NULL, width = NULL, disabled = FALSE, ...)  
actionLink(inputId, label, icon = NULL, ...)```

Arguments

**inputId**
The input slot that will be used to access the value.

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

**icon**
An optional `icon()` to appear on the button.

**width**
The width of the input, e.g. '400px', or '100%'; see `validateCssUnit()`.

**disabled**
If TRUE, the button will not be clickable. Use `updateActionButton()` to dynamically enable/disable the button.

... Named attributes to be applied to the button or link.

Server value

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

- Event handlers (e.g., `observeEvent()`, `eventReactive()`) won’t execute on initial load.
- Input validation (e.g., `req()`, `need()`) will fail on initial load.

See Also

`observeEvent()` and `eventReactive()`

Other input elements: `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `numericInput()`, `passwordInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, `submitButton()`, `textField()`, `varSelectInput()`

Examples

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

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

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

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

  shinyApp(ui, server)
```
## Example of adding extra class values

```r
actionButton("largeButton", "Large Primary Button", class = "btn-primary btn-lg")
actionLink("infoLink", "Information Link", class = "btn-info")
```

### Description

Add, remove, or list directory of static resources to Shiny’s web server, with the given path prefix. Primarily intended for package authors to make supporting JavaScript/CSS files available to their components.

### Usage

```r
addResourcePath(prefix, directoryPath)
resourcePaths()
removeResourcePath(prefix)
```

### Arguments

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

- **directoryPath**
  - The directory that contains the static resources to be served.

### Details

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

1. Static files under the `www/` directory are automatically made available under a request path that begins with `/`.
2. `addResourcePath()` makes static files in a `directoryPath` available under a request path that begins with `prefix`.

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

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

- `addResourcePath()` adds a directory of static resources.
- `resourcePaths()` lists the currently active resource mappings.
- `removeResourcePath()` removes a directory of static resources.
bindCache

Add caching with reactivity to an object

Description

bindCache() adds caching reactive() expressions and render* functions (like renderText(), renderTable(), ...).

Ordinary reactive() expressions automatically cache their most recent value, which helps to avoid redundant computation in downstream reactives. bindCache() will cache all previous values (as long as they fit in the cache) and they can be shared across user sessions. This allows bindCache() to dramatically improve performance when used correctly.

Usage

bindCache(x, ..., cache = "app")

Arguments

x The object to add caching to.

... One or more expressions to use in the caching key.

cache The scope of the cache, or a cache object. This can be "app" (the default), "session", or a cache object like a cachem::cache_disk(). See the Cache Scoping section for more information.

Details

bindCache() requires one or more expressions that are used to generate a cache key, which is used to determine if a computation has occurred before and hence can be retrieved from the cache. If you're familiar with the concept of memoizing pure functions (e.g., the memoise package), you can think of the cache key as the input(s) to a pure function. As such, one should take care to make sure the use of bindCache() is pure in the same sense, namely:

1. For a given key, the return value is always the same.
2. Evaluation has no side-effects.

See Also

singleton()

Examples

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

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

bindCache

Add caching with reactivity to an object

Description

bindCache() adds caching reactive() expressions and render* functions (like renderText(), renderTable(), ...).

Ordinary reactive() expressions automatically cache their most recent value, which helps to avoid redundant computation in downstream reactives. bindCache() will cache all previous values (as long as they fit in the cache) and they can be shared across user sessions. This allows bindCache() to dramatically improve performance when used correctly.

Usage

bindCache(x, ..., cache = "app")

Arguments

x The object to add caching to.

... One or more expressions to use in the caching key.

cache The scope of the cache, or a cache object. This can be "app" (the default), "session", or a cache object like a cachem::cache_disk(). See the Cache Scoping section for more information.

Details

bindCache() requires one or more expressions that are used to generate a cache key, which is used to determine if a computation has occurred before and hence can be retrieved from the cache. If you're familiar with the concept of memoizing pure functions (e.g., the memoise package), you can think of the cache key as the input(s) to a pure function. As such, one should take care to make sure the use of bindCache() is pure in the same sense, namely:

1. For a given key, the return value is always the same.
2. Evaluation has no side-effects.
In the example here, the bindCache() key consists of input$x and input$y combined, and the value is input$x * input$y. In this simple example, for any given key, there is only one possible returned value.

\[
r <- \text{reactive}([[\text{input$x \times input$y}}]) >\%
\text{bindCache(input$x, input$y)}
\]

The largest performance improvements occur when the cache key is fast to compute and the reactive expression is slow to compute. To see if the value should be computed, a cached reactive evaluates the key, and then serializes and hashes the result. If the resulting hashed key is in the cache, then the cached reactive simply retrieves the previously calculated value and returns it; if not, then the value is computed and the result is stored in the cache before being returned.

To compute the cache key, bindCache() hashes the contents of ..., so it's best to avoid including large objects in a cache key since that can result in slow hashing. It's also best to avoid reference objects like environments and R6 objects, since the serialization of these objects may not capture relevant changes.

If you want to use a large object as part of a cache key, it may make sense to do some sort of reduction on the data that still captures information about whether a value can be retrieved from the cache. For example, if you have a large data set with timestamps, it might make sense to extract the most recent timestamp and return that. Then, instead of hashing the entire data object, the cached reactive only needs to hash the timestamp.

\[
r <- \text{reactive}([[\text{compute(bigdata())}}]) >\%
\text{bindCache([[\text{extract_most_recent_time(bigdata())}}])}
\]

For computations that are very slow, it often makes sense to pair bindCache() with bindEvent() so that no computation is performed until the user explicitly requests it (for more, see the Details section of bindEvent()).

**Cache keys and reactivity**

Because the value expression (from the original reactive()) is cached, it is not necessarily re-executed when someone retrieves a value, and therefore it can't be used to decide what objects to take reactive dependencies on. Instead, the key is used to figure out which objects to take reactive dependencies on. In short, the key expression is reactive, and value expression is no longer reactive.

Here's an example of what not to do: if the key is input$x and the value expression is from reactive((input$x + input$y)), then the resulting cached reactive will only take a reactive dependency on input$x – it won't recompute {input$x + input$y} when just input$y changes. Moreover, the cache won't use input$y as part of the key, and so it could return incorrect values in the future when it retrieves values from the cache. (See the examples below for an example of this.)

A better cache key would be something like input$x, input$y. This does two things: it ensures that a reactive dependency is taken on both input$x and input$y, and it also makes sure that both values are represented in the cache key.

In general, key should use the same reactive inputs as value, but the computation should be simpler. If there are other (non-reactive) values that are consumed, such as external data sources, they should be used in the key as well. Note that if the key is large, it can make sense to do some sort of reduction on it so that the serialization and hashing of the cache key is not too expensive.
Remember that the key is reactive, so it is not re-executed every single time that someone accesses the cached reactive. It is only re-executed if it has been invalidated by one of the reactives it depends on. For example, suppose we have this cached reactive:

```r
r <- reactive({ input$x * input$y }) %>%
  bindCache(input$x, input$y)
```

In this case, the key expression is essentially `reactive(list(input$x, input$y))` (there’s a bit more to it, but that’s a good enough approximation). The first time `r()` is called, it executes the key, then fails to find it in the cache, so it executes the value expression, `{ input$x + input$y }`. If `r()` is called again, then it does not need to re-execute the key expression, because it has not been invalidated via a change to `input$x` or `input$y`; it simply returns the previous value. However, if `input$x` or `input$y` changes, then the reactive expression will be invalidated, and the next time that someone calls `r()`, the key expression will need to be re-executed.

Note that if the cached reactive is passed to `bindEvent()`, then the key expression will no longer be reactive; instead, the event expression will be reactive.

**Cache scope**

By default, when `bindCache()` is used, it is scoped to the running application. That means that it shares a cache with all user sessions connected to the application (within the R process). This is done with the cache parameter’s default value, "app".

With an app-level cache scope, one user can benefit from the work done for another user’s session. In most cases, this is the best way to get performance improvements from caching. However, in some cases, this could leak information between sessions. For example, if the cache key does not fully encompass the inputs used by the value, then data could leak between the sessions. Or if a user sees that a cached reactive returns its value very quickly, they may be able to infer that someone else has already used it with the same values.

It is also possible to scope the cache to the session, with `cache="session"`. This removes the risk of information leaking between sessions, but then one session cannot benefit from computations performed in another session.

It is possible to pass in caching objects directly to `bindCache()`. This can be useful if, for example, you want to use a particular type of cache with specific cached reactives, or if you want to use a `cachem::cache_disk()` that is shared across multiple processes and persists beyond the current R session.

To use different settings for an application-scoped cache, you can call `shinyOptions()` at the top of your app.R, server.R, or global.R. For example, this will create a cache with 500 MB of space instead of the default 200 MB:

```r
shinyOptions(cache = cachem::cache_mem(max_size = 500e6))
```

To use different settings for a session-scoped cache, you can set `session$cache` at the top of your server function. By default, it will create a 200 MB memory cache for each session, but you can replace it with something different. To use the session-scoped cache, you must also call `bindCache()` with `cache="session"`. This will create a 100 MB cache for the session:
function(input, output, session) {
  session$cache <- cachem::cache_mem(max_size = 100e6)
  ...
}

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

```r
shinyOptions(cache = cachem::cache_disk(file.path(dirname(tempdir()), "myapp-cache"))
```

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

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

```r
shinyOptions(cache = cachem::cache_disk("./myapp-cache"))
```

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

You can also scope a cache to just one item, or selected items. To do that, create a `cachem::cache_mem()` or `cachem::cache_disk()`, and pass it as the `cache` argument of `bindCache()`.

### Computing cache keys

The actual cache key that is used internally takes value from evaluating the key expression(s) (from the ... arguments) and combines it with the (unevaluated) value expression.

This means that if there are two cached reactives which have the same result from evaluating the key, but different value expressions, then they will not need to worry about collisions.

However, if two cached reactives have identical key and value expressions expressions, they will share the cached values. This is useful when using `cache="app"`: there may be multiple user sessions which create separate cached reactive objects (because they are created from the same code in the server function, but the server function is executed once for each user session), and those cached reactive objects across sessions can share values in the cache.

### Async with cached reactives

With a cached reactive expression, the key and/or value expression can be asynchronous. In other words, they can be promises — not regular R promises, but rather objects provided by the `promises` package, which are similar to promises in JavaScript. (See `promises::promise()` for more information.) You can also use `future::future()` objects to run code in a separate process or even on a remote machine.

If the value returns a promise, then anything that consumes the cached reactive must expect it to return a promise.

Similarly, if the key is a promise (in other words, if it is asynchronous), then the entire cached reactive must be asynchronous, since the key must be computed asynchronously before it knows whether to compute the value or the value is retrieved from the cache. Anything that consumes the cached reactive must therefore expect it to return a promise.
Developing render functions for caching

If you’ve implemented your own `render*()` function, it may just work with `bindCache()`, but it is possible that you will need to make some modifications. These modifications involve helping `bindCache()` avoid cache collisions, dealing with internal state that may be set by the, `render` function, and modifying the data as it goes in and comes out of the cache.

You may need to provide a cache hint to `createRenderFunction()` (or `htmlwidgets::shinyRenderWidget()`, if you’ve authored an htmlwidget) in order for `bindCache()` to correctly compute a cache key.

The potential problem is a cache collision. Consider the following:

```r
output$x1 <- renderText({ input$x }) %>% bindCache(input$x)
output$x2 <- renderText({ input$x * 2 }) %>% bindCache(input$x)
```

Both `output$x1` and `output$x2` use `input$x` as part of their cache key, but if it were the only thing used in the cache key, then the two outputs would have a cache collision, and they would have the same output. To avoid this, a cache hint is automatically added when `renderText()` calls `createRenderFunction()`. The cache hint is used as part of the actual cache key, in addition to the one passed to `bindCache()` by the user. The cache hint can be viewed by calling the internal Shiny function `extractCacheHint()`:

```r
r <- renderText({ input$x })
shiny:::extractCacheHint(r)
```

This returns a nested list containing an item, `$origUserFunc$body`, which in this case is the expression which was passed to `renderText()`: `{ input$x }`. This (quoted) expression is mixed into the actual cache key, and it is how `output$x1` does not have collisions with `output$x2`.

For most developers of render functions, nothing extra needs to be done; the automatic inference of the cache hint is sufficient. Again, you can check it by calling `shiny:::extractCacheHint()`, and by testing the render function for cache collisions in a real application.

In some cases, however, the automatic cache hint inference is not sufficient, and it is necessary to provide a cache hint. This is true for `renderPrint()`. Unlike `renderText()`, it wraps the user-provided expression in another function, before passing it to `createRenderFunction()` (instead of `createRenderFunction()`). Because the user code is wrapped in another function, `createRenderFunction()` is not able to automatically extract the user-provided code and use it in the cache key. Instead, `renderPrint` calls `createRenderFunction()`, it explicitly passes along a cache hint, which includes a label and the original user expression.

In general, if you need to provide a cache hint, it is best practice to provide a label id, the user’s expr, as well as any other arguments that may influence the final value.

For `htmlwidgets`, it will try to automatically infer a cache hint; again, you can inspect the cache hint with `shiny:::extractCacheHint()` and also test it in an application. If you do need to explicitly provide a cache hint, pass it to `shinyRenderWidget`. For example:

```r
renderMyWidget <- function(expr) {
  q <- rlang::enquo0(expr)

  htmlwidgets::shinyRenderWidget(
```
If your render function sets any internal state, you may find it useful in your call to `createRenderFunction()` to use the cacheWriteHook and/or cacheReadHook parameters. These hooks are functions that run just before the object is stored in the cache, and just after the object is retrieved from the cache. They can modify the data that is stored and retrieved; this can be useful if extra information needs to be stored in the cache. They can also be used to modify the state of the application; for example, it can call `createWebDependency()` to make JS/CSS resources available if the cached object is loaded in a different R process. (See the source of htmlwidgets::shinyRenderWidget for an example of this.)

**Uncacheable objects**

Some render functions cannot be cached, typically because they have side effects or modify some external state, and they must re-execute each time in order to work properly.

For developers of such code, they should call `createRenderFunction()` (or `markRenderFunction()`) with cacheHint = FALSE.

**Caching with `renderPlot()`**

When `bindCache()` is used with `renderPlot()`, the height and width passed to the original `renderPlot()` are ignored. They are superseded by `sizePolicy` argument passed to `bindCache`. The default is:

```
sizePolicy = sizeGrowthRatio(width = 400, height = 400, growthRate = 1.2)
```

`sizePolicy` must be a function that takes a two-element numeric vector as input, representing the width and height of the `<img>` element in the browser window, and it must return a two-element numeric vector, representing the pixel dimensions of the plot to generate. The purpose is to round the actual pixel dimensions from the browser to some other dimensions, so that this will not generate and cache images of every possible pixel dimension. See `sizeGrowthRatio()` for more information on the default sizing policy.

**See Also**

`bindEvent()`, `renderCachedPlot()` for caching plots.

**Examples**

```r
## Not run:
rc <- bindCache(
  x = reactive(
    Sys.sleep(2)  # Pretend this is expensive
    input$x * 100
  )
)```
bindCache

## End(Not run)

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

# Basic example
shinyApp(
    ui = fluidPage(
        sliderInput("x", "x", 1, 10, 5),
        sliderInput("y", "y", 1, 10, 5),
        div("x * y: "),
        verbatimTextOutput("txt")
    ),
    server = function(input, output) {
        r <- reactive(
            # The value expression is an _expensive_ computation
            message("Doing expensive computation...")
            Sys.sleep(2)
            input$x * input$y
        )
        bindCache(input$x, input$y)

        output$txt <- renderText(r())
    }
)

# Caching renderText
shinyApp(
    ui = fluidPage(
        sliderInput("x", "x", 1, 10, 5),
        sliderInput("y", "y", 1, 10, 5),
        div("x * y: "),
        verbatimTextOutput("txt")
    ),
    server = function(input, output) {
        output$txt <- renderText({
            message("Doing expensive computation...")
            Sys.sleep(2)
        })
    }
)
input$x * input$y
}) %>%
bindCache(input$x, input$y)
}

# Demo of using events and caching with an actionButton
shinyApp(
  ui = fluidPage(
    sliderInput("x", "x", 1L, 10L, 5L),
    sliderInput("y", "y", 1L, 10L, 5L),
    actionButton("go", "Go"),
    div("x * y: "),
    verbatimTextOutput("txt")
  ),
  server = function(input, output) {
    r <- reactive({
      message("Doing expensive computation...")
      Sys.sleep(2)
      input$x * input$y
    }) %>%
    bindCache(input$x, input$y) %>%
    bindEvent(input$go)
    # The cached, eventified reactive takes a reactive dependency on
    # input$go, but doesn't use it for the cache key. It uses input$x and
    # input$y for the cache key, but doesn't take a reactive dependency on
    # them, because the reactive dependency is superseded by addEvent().

    output$txt <- renderText(r())
  }
)
)

bindEvent

Make an object respond only to specified reactive events

Description

Modify an object to respond to "event-like" reactive inputs, values, and expressions. `bindEvent()`
can be used with reactive expressions, render functions, and observers. The resulting object takes
a reactive dependency on the ... arguments, and not on the original object's code. This can, for
example, be used to make an observer execute only when a button is pressed.

`bindEvent()` was added in Shiny 1.6.0. When it is used with `reactive()` and `observe()`, it
does the same thing as `eventReactive()` and `observeEvent()`. However, `bindEvent()` is more
flexible: it can be combined with `bindCache()`, and it can also be used with render functions (like
`renderText()` and `renderPlot()`).
Usage

```r
bindEvent(
  x,
  ..., 
  ignoreNULL = TRUE,
  ignoreInit = FALSE,
  once = FALSE,
  label = NULL
)
```

Arguments

- **x**: An object to wrap so that is triggered only when a the specified event occurs.
- **...**: One or more expressions 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. If there are multiple expressions in the ... , then it will take a dependency on all of them.
- **ignoreNULL**: Whether the action should be triggered (or value calculated) when the input is `NULL`. See Details.
- **ignoreInit**: If `TRUE`, then, when the eventified object is first created/initialized, don’t trigger the action (or compute the value). The default is `FALSE`. See Details.
- **once**: Used only for observers. Whether this observer should be immediately destroyed after the first time that the code in the observer is run. This pattern is useful when you want to subscribe to a event that should only happen once.
- **label**: A label for the observer or reactive, useful for debugging.

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()`. `bindEvent()` provides a straightforward API for event handling that wraps `observe` and `isolate`.

The ... arguments are captured as expressions and combined into an event expression. When this event expression is invalidated (when its upstream reactive inputs change), that is an event, and it will cause the original object’s code to execute.

Use `bindEvent()` with `observe()` whenever you want to perform an action in response to an event. (This does the same thing as `observeEvent()`, which was available in Shiny prior to version 1.6.0.) Note that "recalculate a value" does not generally count as performing an action – use `reactive()` for that.

Use `bindEvent()` with `reactive()` 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. (This does the same thing as `eventReactive()`, which was available in Shiny prior to version 1.6.0.)

`bindEvent()` is often used with `bindCache()`.

**ignoreNULL and ignoreInit**

`bindEvent()` takes an `ignoreNULL` parameter that affects behavior when the event expression evaluates to NULL (or in the special case of an `actionButton()`, 0). In these cases, if `ignoreNULL` is TRUE, then it 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).

`bindEvent()` also takes an `ignoreInit` argument. By default, reactive expressions and observers will run on the first reactive flush after they are created (except if, at that moment, the event expression evaluates to NULL and `ignoreNULL` is TRUE). But when responding to a click of an action button, it may often be useful to set `ignoreInit` to TRUE. For example, if you’re setting up an observer to respond to a dynamically created button, then `ignoreInit = TRUE` will guarantee that the action will only be triggered when the button is actually clicked, instead of also being triggered when it is created/initialized. Similarly, if you’re setting up a reactive that responds to a dynamically created button used to refresh some data (which is then returned by that reactive), then you should use `reactive(...) %>% bindEvent(..., ignoreInit = TRUE)` if you want to let the user decide if/when they want to refresh the data (since, depending on the app, this may be a computationally expensive operation).

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

- **ignoreNULL = TRUE and ignoreInit = FALSE**  This is the default. This combination means that reactive/observer code will run every time that event expression is not NULL. If, at the time of creation, the event expression happens to not be NULL, then the code runs.

- **ignoreNULL = FALSE and ignoreInit = FALSE**  This combination means that reactive/observer code will run every time no matter what.

- **ignoreNULL = FALSE and ignoreInit = TRUE**  This combination means that reactive/observer code will not run at the time of creation (because `ignoreInit = TRUE`), but it will run every other time.

- **ignoreNULL = TRUE and ignoreInit = TRUE**  This combination means that reactive/observer code will not at the time of creation (because `ignoreInit = TRUE`). After that, the reactive/observer code will run every time that the event expression is not NULL.

**Types of objects**

`bindEvent()` can be used with reactive expressions, observers, and shiny render functions.

When `bindEvent()` is used with `reactive()`, it creates a new reactive expression object.

When `bindEvent()` is used with `observe()`, it alters the observer in place. It can only be used with observers which have not yet executed.
Combining events and caching

In many cases, it makes sense to use `bindEvent()` along with `bindCache()`, because they each can reduce the amount of work done on the server. For example, you could have `sliderInputs x` and `y` and a `reactive()` that performs a time-consuming operation with those values. Using `bindCache()` can speed things up, especially if there are multiple users. But it might make sense to also not do the computation until the user sets both `x` and `y`, and then clicks on an `actionButton` named `go`.

To use both caching and events, the object should first be passed to `bindCache()`, then `bindEvent()`. For example:

```r
r <- reactive({
  Sys.sleep(2)  # Pretend this is an expensive computation
  input$x * input$y
}) %>%
bindCache(input$x, input$y) %>%
bindEvent(input$go)
```

Anything that consumes `r()` will take a reactive dependency on the event expression given to `bindEvent()`, and not the cache key expression given to `bindCache()`. In this case, it is just `input$go`.

---

**bookmarkButton**

Create a button for bookmarking/sharing

**Description**

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

**Usage**

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

**Arguments**

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

- **icon**: An optional `icon()` to appear on the button.

- **title**: A tooltip that is shown when the mouse cursor hovers over the button.

- **...**: Named attributes to be applied to the button or link.
id  

An ID for the bookmark button. The only time it is necessary to set the ID unless you have more than one bookmark button in your application. If you specify an input ID, it should be excluded from bookmarking with `setBookmarkExclude()`, and you must create an observer that does the bookmarking when the button is pressed. See the examples below.

**See Also**

`enableBookmarking()` for more examples.

**Examples**

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

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

  # Trigger bookmarking with either button
  observeEvent(input$bookmark1, {
    session$doBookmark()
  })
  observeEvent(input$bookmark2, {
    session$doBookmark()
  })
}

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

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

Usage

bootstrapLib(theme = NULL)

Arguments

theme One of the following:

• NULL (the default), which implies a "stock" build of Bootstrap 3.
• A bslib::bs_theme() object. This can be used to replace a stock build of Bootstrap 3 with a customized version of Bootstrap 3 or higher.
• A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css).

Details

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

bootstrapPage Create a Bootstrap page

Description

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

Usage

bootstrapPage(..., title = NULL, theme = NULL, lang = NULL)

basicPage(...)
• A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css).

lang ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

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 definition 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()

Description

brushedPoints() returns rows from a data frame which are under a brush. nearPoints() returns rows from a data frame which are near a click, hover, or double-click. Alternatively, set allRows = TRUE to return all rows from the input data with an additional column selected_ that indicates which rows of the would be selected.

Usage

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

nearPoints(

brushedPoints

def, coordinfo, xvar = NULL, yvar = NULL, panelvar1 = NULL, panelvar2 = NULL, threshold = 5, maxpoints = NULL, addDist = FALSE, allRows = FALSE

Arguments

df A data frame from which to select rows.

brush, coordinfo The data from a brush or click/dblclick/hover event e.g. input$plot_brush, input$plot_click.

xvar, yvar A string giving the name of the variable on the x or y axis. These are only required for base graphics, and must be the name of a column in df.

panelvar1, panelvar2 A string giving the name of a panel variable. For expert use only; in most cases these will be automatically derived from the ggplot2 spec.

allRows If FALSE (the default) return a data frame containing the selected rows. If TRUE, the input data frame will have a new column, selected_, which indicates whether the row was selected or not.

threshold A maximum distance (in pixels) to the pointer location. Rows in the data frame will be selected if the distance to the pointer is less than threshold.

maxpoints Maximum number of rows to return. If NULL (the default), will return all rows within the threshold distance.

addDist If TRUE, add a column named dist_ that contains the distance from the coordinate to the point, in pixels. When no pointer event has yet occurred, the value of dist_ will be NA.

Value

A data frame based on df, containing the observations selected by the brush or near the click event. For nearPoints(), the rows will be sorted by distance to the event.

If allRows = TRUE, then all rows will returned, along with a new selected_ column that indicates whether or not the point was selected. The output from nearPoints() will no longer be sorted, but you can set addDist = TRUE to get an additional column that gives the pixel distance to the pointer.

ggplot2

For plots created with ggplot2, it is not necessary to specify the column names to xvar, yvar, panelvar1, and panelvar2 as that information can be automatically derived from the plot specification.
Note, however, that this will not work if you use a computed column, like `aes(speed/2, dist))`. Instead, we recommend that you modify the data first, and then make the plot with "raw" columns in the modified data.

### Brushing

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

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

### See Also

`plotOutput()` for example usage.

### Examples

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

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

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

### Description

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

### Usage

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

Arguments

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

fill Fill color of the brush. If 'auto', it derives from the link color of the plot's HTML container (if thematic is enabled, and accent is a non-'auto' value, that color is used instead).

stroke Outline color of the brush. If 'auto', it derives from the foreground color of the plot's HTML container (if thematic is enabled, and fg is a non-'auto' value, that color is used instead).

opacity Opacity of the brush

delay How long to delay (in milliseconds) when debouncing or throttling, before sending the brush data to the server.

delayType The type of algorithm for limiting the number of brush events. Use "throttle" to limit the number of brush events to one every delay milliseconds. Use "debounce" to suspend events while the cursor is moving, and wait until the cursor has been at rest for delay milliseconds before sending an event.

clip Should the brush area be clipped to the plotting area? If FALSE, then the user will be able to brush outside the plotting area, as long as it is still inside the image.

direction The direction for brushing. If "xy", the brush can be drawn and moved in both x and y directions. If "x", or "y", the brush will work horizontally or vertically.

resetOnNew When a new image is sent to the browser (via renderImage()), should the brush be reset? The default, FALSE, is useful if you want to update the plot while keeping the brush. Using TRUE is useful if you want to clear the brush whenever the plot is updated.

See Also

clickOpts() for clicking events.
**busyIndicatorOptions**

**Description**

Shiny automatically includes busy indicators, which more specifically means:

1. Calculating/recalculating outputs have a spinner overlay.
2. Outputs fade out/in when recalculating.
3. When no outputs are calculating/recalculating, but Shiny is busy doing something else (e.g., a download, side-effect, etc), a page-level pulsing banner is shown.

This function allows you to customize the appearance of these busy indicators by including the result of this function inside the app’s UI. Note that, unless `spinner_selector` (or `fade_selector`) is specified, the spinner/fade customization applies to the parent element. If the customization should instead apply to the entire page, set `spinner_selector = 'html'` and `fade_selector = 'html'`.

**Usage**

```r
busyIndicatorOptions(
  ..., 
  spinner_type = NULL, 
  spinner_color = NULL, 
  spinner_size = NULL, 
  spinner_delay = NULL, 
  spinner_selector = NULL, 
  fade_opacity = NULL, 
  fade_selector = NULL, 
  pulse_background = NULL, 
  pulse_height = NULL, 
  pulse_speed = NULL
)
```

**Arguments**

```r
... 
spiner_type      Currently ignored.
A path to a local SVG file can also be provided. The SVG should adhere to the following rules:
  • The SVG itself should contain the animation.
  • It should avoid absolute sizes (the spinner’s containing DOM element size is set in CSS by spinner_size, so it should fill that container).
  • It should avoid setting absolute colors (the spinner’s containing DOM element color is set in CSS by spinner_color, so it should inherit that color).
spiner_color     The color of the spinner. This can be any valid CSS color. Defaults to the app’s “primary” color if Bootstrap is on the page.
spiner_size      The size of the spinner. This can be any valid CSS size.
spiner_delay     The amount of time to wait before showing the spinner. This can be any valid CSS time and can be useful for not showing the spinner if the computation finishes quickly.
```
spinner_selector
A character string containing a CSS selector for scoping the spinner customization. The default (NULL) will apply the spinner customization to the parent element of the spinner.

fade_opacity
The opacity (a number between 0 and 1) for recalculating output. Set to 1 to "disable" the fade.

fade_selector
A character string containing a CSS selector for scoping the spinner customization. The default (NULL) will apply the spinner customization to the parent element of the spinner.

pulse_background
A CSS background definition for the pulse. The default uses a linear-gradient of the theme’s indigo, purple, and pink colors.

pulse_height
The height of the pulsing banner. This can be any valid CSS size.

pulse_speed
The speed of the pulsing banner. This can be any valid CSS time.

See Also
useBusyIndicators() to disable/enable busy indicators.

Examples

library(bslib)

card_ui <- function(id, spinner_type = id) {
  card(
    busyIndicatorOptions(spinner_type = spinner_type),
    card_header(paste("Spinner:", spinner_type)),
    plotOutput(shiny::NS(id, "plot"))
  )
}

card_server <- function(id, simulate = reactive()) {
  moduleServer(
    id = id,
    function(input, output, session) {
      output$plot <- renderPlot({
        Sys.sleep(1)
        simulate()
        plot(x = rnorm(100), y = rnorm(100))
      })
    }
  )
}

ui <- page_fillable(
  useBusyIndicators(),
  input_task_button("simulate", "Simulate", icon = icon("refresh")),
  layout_columns(
    card_ui("ring"),
callModule

   card_ui("bars"),
   card_ui("dots"),
   card_ui("pulse"),
   col_widths = 6

server <- function(input, output, session) {
  simulate <- reactive(input$simulate)
  card_server("ring", simulate)
  card_server("bars", simulate)
  card_server("dots", simulate)
  card_server("pulse", simulate)
}

shinyApp(ui, server)

---

callModule

Invoke a Shiny module

---

Description

Note: As of Shiny 1.5.0, we recommend using moduleServer() instead of callModule(), because the syntax is a little easier to understand, and modules created with moduleServer can be tested with testServer().

Usage

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

Arguments

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

Value

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

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

Arguments

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

Value

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

Server value

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

See Also

checkboxInput(), updateCheckboxGroupInput()

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

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    checkboxGroupInput("variable", "Variables to show:",
      c("Cylinders" = "cyl",
        "Transmission" = "am",
        "Gears" = "gear"),
    tableOutput("data")
  )

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

  shinyApp(ui, server)
}

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

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

shinyApp(ui, server)
```
checkboxInput

Checkbox Input Control

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

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

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

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

Server value
TRUE if checked, FALSE otherwise.

See Also
checkboxGroupInput(), updateCheckboxInput()

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

Examples
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    checkboxInput("somevalue", "Some value", FALSE),
    verbatimTextOutput("value")
  )
  server <- function(input, output) {
    output$size <- renderText({ input$somevalue })
  }
  shinyApp(ui, server)
}
clickOpts

Description

These functions give control over the click, dblClick and hover events generated by imageOutput() and plotOutput().

Usage

```r
clickOpts(id, clip = TRUE)
dblclickOpts(id, clip = TRUE, delay = 400)
hoverOpts(
  id,
  delay = 300,
  delayType = c("debounce", "throttle"),
  clip = TRUE,
  nullOutside = TRUE
)
```

Arguments

- **id**  
  Input value name. For example, if the value is "plot_click", then the event data will be available as input$plot_click.

- **clip**  
  Should the click area be clipped to the plotting area? If FALSE, then the server will receive click events even when the mouse is outside the plotting area, as long as it is still inside the image.

- **delay**  
  For dblClickOpts(): the maximum delay (in ms) between a pair clicks for them to be counted as a double-click.
  
  For hoverOpts(): how long to delay (in ms) when debouncing or throttling before sending the mouse location to the server.

- **delayType**  
  The type of algorithm for limiting the number of hover events. Use "throttle" to limit the number of hover events to one every delay milliseconds. Use "debounce" to suspend events while the cursor is moving, and wait until the cursor has been at rest for delay milliseconds before sending an event.

- **nullOutside**  
  If TRUE (the default), the value will be set to NULL when the mouse exits the plotting area. If FALSE, the value will stop changing when the cursor exits the plotting area.

See Also

- brushOpts() for brushing events.
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

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

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

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

ui <- fluidPage(
  fluidRow(
    column(width = 4, "4"),
    column(width = 3, offset = 2, "3 offset 2")
  )
)

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

conditionalPanel

Description

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

Usage

conditionalPanel(condition, ..., ns = NS(NULL))

Arguments

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

Details

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

Note

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

```r
## Only run this example in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    sidebarPanel(
      selectInput("plotType", "Plot Type",
        c(Scatter = "scatter", Histogram = "hist")
    ),
    # Only show this panel if the plot type is a histogram
    conditionalPanel(
      condition = "input.plotType == 'hist'",
      selectInput("breaks", "Breaks",
        c("Sturges", "Scott", "Freedman-Diaconis", "[Custom]" = "custom")
      ),
      # Only show this panel if Custom is selected
      conditionalPanel(
        condition = "input.breaks == 'custom'",
        sliderInput("breakCount", "Break Count", min = 1, max = 50, value = 10)
      )
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
  server <- function(input, output) {
    x <- rnorm(100)
    y <- rnorm(100)

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

        hist(x, breaks = breaks)
      }
    })

    shinyApp(ui, server)
  }
```

createRenderFunction  Implement custom render functions
createRenderFunction

Description

Developer-facing utilities for implementing a custom renderXXX() function. Before using these utilities directly, consider using the htmlwidgets package to implement custom outputs (i.e., custom renderXXX()/xxxOutput() functions). That said, these utilities can be used more directly if a full-blown htmlwidget isn’t needed and/or the user-supplied reactive expression needs to be wrapped in additional call(s).

Usage

createRenderFunction(
  func,
  transform = function(value, session, name, ...) value,
  outputFunc = NULL,
  outputArgs = NULL,
  cacheHint = "auto",
  cacheWriteHook = NULL,
  cacheReadHook = NULL
)

quoToFunction(q, label = sys.call(-1)[[1]], ..stacktraceon = FALSE)

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

Arguments

- **func**: A function without parameters, that returns user data. If the returned value is a promise, then the render function will proceed in async mode.
- **transform**: A function that takes four arguments: value, session, name, and ... (for future-proofing). This function will be invoked each time a value is returned from func, and is responsible for changing the value into a JSON-ready value to be JSON-encoded and sent to the browser.
- **outputFunc**: The UI function that is used (or most commonly used) with this render function. This can be used in R Markdown documents to create complete output widgets out of just the render function.
- **outputArgs**: A list of arguments to pass to the uiFunc. Render functions should include outputArgs = list() in their own parameter list, and pass through the value to markRenderFunction, to allow app authors to customize outputs. (Currently, this is only supported for dynamically generated UIs, such as those created by Shiny code snippets embedded in R Markdown documents).
cacheHint

One of "auto", FALSE, or some other information to identify this instance for caching using bindCache(). If "auto", it will try to automatically infer caching information. If FALSE, do not allow caching for the object. Some render functions (such as renderPlot) contain internal state that makes them unsuitable for caching.

cacheWriteHook

Used if the render function is passed to bindCache(). This is an optional callback function to invoke before saving the value from the render function to the cache. This function must accept one argument, the value returned from renderFunc, and should return the value to store in the cache.

cacheReadHook

Used if the render function is passed to bindCache(). This is an optional callback function to invoke after reading a value from the cache (if there is a cache hit). The function will be passed one argument, the value retrieved from the cache. This can be useful when some side effect needs to occur for a render function to behave correctly. For example, some render functions call createWebDependency() so that Shiny is able to serve JS and CSS resources.

q

Quosure of the expression x. When capturing expressions to create your quosure, it is recommended to use rlang::enquo() to not unquote the object too early. See rlang::enquo() for more details.

label

A label for the object to be shown in the debugger. Defaults to the name of the calling function.

expr

A quoted or unquoted expression, or a quosure.

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.

wrappedWithLabel, ...stacktraceon

Advanced use only. For stack manipulation purposes; see stacktrace().

Details

To implement a custom renderXXX() function, essentially 2 things are needed:

1. Capture the user’s reactive expression as a function.
   
   • New renderXXX() functions can use quoToFunction() for this, but already existing renderXXX() functions that contain env and quoted parameters may want to continue using installExprFunction() for better legacy support (see examples).

2. Flag the resulting function (from 1) as a Shiny rendering function and also provide a UI container for displaying the result of the rendering function.
   
   • createRenderFunction() is currently recommended (instead of markRenderFunction()) for this step (see examples).

Value

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

- `quoToFunction()`: convert a quosure to a function.
- `installExprFunction()`: converts a user's reactive `expr` into a function that's assigned to a name in the `assign.env`.

**Examples**

```r
# A custom render function that repeats the supplied value 3 times
renderTriple <- function(expr) {
  # Wrap user-supplied reactive expression into a function
  func <- quoToFunction(rlang::enquo0(expr))

  createRenderFunction(
    func,
    transform = function(value, session, name, ...) {
      paste(rep(value, 3), collapse=",
    },
    outputFunc = textOutput
  )
}

# For better legacy support, consider using `installExprFunction()` over `quoToFunction()`
renderTripleLegacy <- function(expr, env = parent.frame(), quoted = FALSE) {
  func <- installExprFunction(expr, "func", env, quoted)

  createRenderFunction(
    func,
    transform = function(value, session, name, ...) {
      paste(rep(value, 3), collapse=",
    },
    outputFunc = textOutput
  )
}

# Test render function from the console
reactiveConsole(TRUE)

v <- reactiveVal("basic")
r <- renderTriple({ v() })
r()
#> [1] "basic, basic, basic"

# User can supply quoted code via `rlang::quo()`. Note that evaluation of the
# expression happens when `r2()` is invoked, not when `r2` is created.
q <- rlang::quo({ v() })
r2 <- rlang::inject(renderTriple(!!q))
v("rlang")
r2()
#> [1] "rlang, rlang, rlang"

# Supplying quoted code without `rlang::quo()` requires `installExprFunction()`
expr <- quote({ v() })
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, scrubFile = TRUE)

Arguments

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

scrubFile: If TRUE (the default), remove src$file for the dependency. This prevents the local file path from being sent to the client when dynamic web dependencies are used. If FALSE, don’t remove src$file. Setting it to FALSE should be needed only in very unusual cases.

Value

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

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

**Usage**

dateInput(
  inputId, 
  label, 
  value = NULL, 
  min = NULL, 
  max = NULL, 
  format = "yyyy-mm-dd", 
  startview = "month", 
  weekstart = 0, 
  language = "en", 
  width = NULL, 
  autoclose = TRUE, 
  datesdisabled = NULL, 
  daysofweekdisabled = NULL
)

**Arguments**

- **inputId**: The input slot that will be used to access the value.
- **label**: Display label for the control, or NULL for no label.
- **value**: The starting date. Either a Date object, or a string in yyyy-mm-dd format. If NULL (the default), will use the current date in the client’s time zone.
- **min**: The minimum allowed date. Either a Date object, or a string in yyyy-mm-dd format.
- **max**: The maximum allowed date. Either a Date object, or a string in yyyy-mm-dd format.
- **format**: The format of the date to display in the browser. Defaults to "yyyy-mm-dd".
- **startview**: The date range shown when the input object is first clicked. Can be "month" (the default), "year", or "decade".
- **weekstart**: Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).
- **width**: The width of the input slot. Defaults to NULL.
- **autoclose**: Automatically closes the calendar after a selection is made. Defaults to TRUE.
- **datesdisabled**: An array of dates that are disabled. Defaults to NULL.
- **daysofweekdisabled**: An array of days that are disabled. Defaults to NULL.

width The width of the input, e.g. '400px', or '100%'; see validateCssUnit().

autoclose Whether or not to close the datepicker immediately when a date is selected.

datesdisabled Which dates should be disabled. Either a Date object, or a string in yyyy-mm-dd format.

daysofweekdisabled Days of the week that should be disabled. Should be a integer vector with values from 0 (Sunday) to 6 (Saturday).

Details

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

- `yy` Year without century (12)
- `yyyy` Year with century (2012)
- `mm` Month number, with leading zero (01-12)
- `m` Month number, without leading zero (1-12)
- `M` Abbreviated month name
- `MM` Full month name
- `dd` Day of month with leading zero
- `d` Day of month without leading zero
- `D` Abbreviated weekday name
- `DD` Full weekday name

Server value

A Date vector of length 1.

See Also

dateRangeInput(), updateDateInput()

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

Examples

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

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

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

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

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

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

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

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

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

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

dateRangeInput Create date range input

Description

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

Usage

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

### Arguments

- **inputId**
The input slot that will be used to access the value.

- **label**
Display label for the control, or NULL for no label.

- **start**
The initial start date. Either a Date object, or a string in `yyyy-mm-dd` format. If NULL (the default), will use the current date in the client’s time zone.

- **end**
The initial end date. Either a Date object, or a string in `yyyy-mm-dd` format. If NULL (the default), will use the current date in the client’s time zone.

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

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

- **format**
The format of the date to display in the browser. Defaults to "yyyy-mm-dd".

- **startview**
The date range shown when the input object is first clicked. Can be "month" (the default), "year", or "decade".

- **weekstart**
Which day is the start of the week. Should be an integer from 0 (Sunday) to 6 (Saturday).

- **language**

- **separator**
String to display between the start and end input boxes.

- **width**
The width of the input, e.g. '400px', or '100%'; see `validateCssUnit()`.

- **autoclose**
Whether or not to close the datepicker immediately when a date is selected.

### Details

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

- **yy** Year without century (12)
- **yyyy** Year with century (2012)
- **mm** Month number, with leading zero (01-12)
- **m** Month number, without leading zero (1-12)
**dateRangeInput**

- M Abbreviated month name
- MM Full month name
- dd Day of month with leading zero
- d Day of month without leading zero
- D Abbreviated weekday name
- DD Full weekday name

**Server value**

A Date vector of length 2.

**See Also**

dateInput(), updateDateRangeInput()

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

**Examples**

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    dateRangeInput("daterange1", "Date range: ",
                      start = "2001-01-01",
                      end = "2010-12-31"),

    dateRangeInput("daterange2", "Date range: ",
                      start = "2001-01-01",
                      end = "2010-12-31",
                      min = "2001-01-01",
                      max = "2012-12-21",
                      format = "mm/dd/yy",
                      separator = " - ").

    dateRangeInput("daterange3", "Date range: ",
                      start = "2001-01-01",
                      end = "2010-12-31",
                      min = "2001-01-01",
                      max = "2012-12-21",
                      format = "mm/dd/yy",
                      separator = " - ").

    dateRangeInput("daterange4", "Date range: ",
                      start = Sys.Date()-10,
                      end = Sys.Date()+10),

    dateRangeInput("daterange5", "Date range: ",
                      language = "de",
                      weekstart = 1),
```
# Start with decade view instead of default month view

dateRangeInput("daterange6", "Date range:",
    startview = "decade")

)

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

debounce

## debounce

Slow down a reactive expression with debounce/throttle

### Description

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

### Usage

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

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

### Arguments

- **r**
  - A reactive expression (that invalidates too often).

- **millis**
  - The debounce/throttle time window. You may optionally pass a no-arg function or reactive expression instead, e.g. to let the end-user control the time window.

- **priority**
  - Debounce/throttle is implemented under the hood using observers. Use this parameter to set the priority of these observers. Generally, this should be higher than the priorities of downstream observers and outputs (which default to zero).

- **domain**
  - See domains.

### Details

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

Debouncing means that every invalidation from r will be held for the specified time window. If r invalidates again within that time window, then the timer starts over again. This means that as long as invalidations continually arrive from r within the time window, the debounced reactive will not
debounce

invalidate at all. Only after the invalidations stop (or slow down sufficiently) will the downstream invalidation be sent.

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

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

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

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

Limitations

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

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

Examples

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

  library(shiny)
  library(magrittr)

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

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

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

Shiny Developer Mode

Description

[Experimental]
Developer Mode enables a number of \code{options()} to make a developer's life easier, like enabling non-minified JS and printing messages about deprecated functions and options.
Shiny Developer Mode can be enabled by calling \code{devmode(TRUE)} and disabled by calling \code{devmode(FALSE)}.
Please see the function descriptions for more details.

Usage

\begin{Schunk}
\begin{Sinput}
devmode(
  devmode = getOption("shiny.devmode", TRUE),
  verbose = getOption("shiny.devmode.verbose", TRUE)
)
\end{Sinput}
\end{Schunk}

\begin{Schunk}
\begin{Sinput}
in_devmode()
\end{Sinput}
\end{Schunk}

\begin{Schunk}
\begin{Sinput}
with_devmode(devmode, code, verbose = getOption("shiny.devmode.verbose", TRUE))
\end{Sinput}
\end{Schunk}

\begin{Schunk}
\begin{Sinput}
devmode_inform(
  message,
  .frequency = "regularly",
  .frequency_id = message,
  .file = stderr(),
  ...
)
\end{Sinput}
\end{Schunk}

\begin{Schunk}
\begin{Sinput}
register_devmode_option(name, devmode_message = NULL, devmode_default = NULL)
\end{Sinput}
\end{Schunk}
get_devmode_option(
    name,        
    default = NULL, 
    devmode_default = missing_arg(), 
    devmode_message = missing_arg()
)

Arguments

- **devmode**: Logical value which should be set to TRUE to enable Shiny Developer Mode
- **verbose**: Logical value which should be set to TRUE to display Shiny Developer messages
- **code**: Code to execute with the temporary Dev Mode options set
- **message**: Developer Mode message to be sent to `rlang::inform()`. Defaults to once every 8 hours.
- **.frequency**: Frequency of the Developer Mode message used with `rlang::inform()`. Defaults to once every 8 hours.
- **.frequency_id**: `rlang::inform()` message identifier. Defaults to `message`.
- **.file**: Output connection for `rlang::inform()`. Defaults to `stderr()`
- **...**: Parameters passed to `rlang::inform()`
- **name**: Name of option to look for in `options()`
- **devmode_message**: Message to display once every 8 hours when utilizing the devmode_default value. If devmode_message is missing, the registered devmode_message value be used.
- **devmode_default**: Default value to return if `in_devmode()` returns TRUE and the specified option is not set in `options()`. For `get_devmode_option()`, if devmode_default is missing, the registered devmode_default value will be used.
- **default**: Default value to return if `in_devmode()` returns TRUE and the specified option is not set in `options()`.

Functions

- **devmode()**: Function to set two options to enable/disble Shiny Developer Mode and Developer messages
- **in_devmode()**: Determines if Shiny is in Developer Mode. If the `getOption("shiny.devmode")` is set to TRUE and not in testing inside testthat, then Shiny Developer Mode is enabled.
- **with_devmode()**: Temporarily set Shiny Developer Mode and Developer message verbosity
- **devmode_inform()**: If Shiny Developer Mode and verbosity are enabled, displays a message once every 8 hrs (by default)
- **register_devmode_option()**: Registers a Shiny Developer Mode option with an updated value and Developer message. This registration method allows package authors to write one message in a single location.

For example, the following Shiny Developer Mode options are registered:
# Reload the Shiny app when a sourced R file changes
register_devmode_option(  
  "shiny.autoreload",  
  "Turning on shiny autoreload. To disable, call `options(shiny.autoreload = FALSE)`",  
  devmode_default = TRUE
)

# Use the unminified Shiny JavaScript file, `shiny.js`
register_devmode_option(  
  "shiny.minified",  
  "Using full shiny javascript file. To use the minified version, call `options(shiny.minified = TRUE)`",  
  devmode_default = FALSE
)

# Display the full stack trace when errors occur during Shiny app execution
register_devmode_option(  
  "shiny.fullstacktrace",  
  "Turning on full stack trace. To disable, call `options(shiny.fullstacktrace = FALSE)`",  
  devmode_default = TRUE
)

Other known, non-Shiny Developer Mode options:

- Sass:

  # Display the full stack trace when errors occur during Shiny app execution
  register_devmode_option(  
    "sass.cache",  
    "Turning off sass cache. To use default caching, call `options(sass.cache = TRUE)`",  
    devmode_default = FALSE
  )

- `get_devmode_option()`: Provides a consistent way to change the expected `getOption()` behavior when Developer Mode is enabled. This method is very similar to `getOption()` where the globally set option takes precedence. See section "Avoiding direct dependency on shiny" for `get_devmode_option()` implementation details.

  **Package developers**: Register your Dev Mode option using `register_devmode_option()` to avoid supplying the same `devmode_default` and `devmode_message` values throughout your package. (This requires a shiny dependency.)

**Avoiding direct dependency on shiny**

The methods explained in this help file act independently from the rest of Shiny but are included to provide blue prints for your own packages. If your package already has (or is willing to take) a dependency on Shiny, we recommend using the exported Shiny methods for consistent behavior. Note that if you use exported Shiny methods, it will cause the Shiny package to load. This may be undesirable if your code will be used in (for example) R Markdown documents that do not have a Shiny runtime (runtime: shiny).

If your package can **not** take a dependency on Shiny, we recommending re-implementing these two functions:
1. in_devmode():
   This function should return TRUE if getOption("shiny.devmode") is set. In addition, we strongly recommend that it also checks to make sure testthat is not testing.

   ```r
   in_devmode <- function() {
     isTRUE(getOption("shiny.devmode", FALSE)) &&
     !identical(Sys.getenv("TESTTHAT"), "true")
   }
   ```

2. get_devmode_option(name, default, devmode_default, devmode_message):
   This function is similar to getOption(name, default), but when the option is not set, the default value changes depending on the Dev Mode. get_devmode_option() should be implemented as follows:
   - If not in Dev Mode:
     - Return getOption(name, default).
   - If in Dev Mode:
     - Get the global option getOption(name) value.
     - If the global option value is set:
       * Return the value.
     - If the global option value is not set:
       * Notify the developer that the Dev Mode default value will be used.
       * Return the Dev Mode default value.

   When notifying the developer that the default value has changed, we strongly recommend displaying a message (devmode_message) to stderr() once every 8 hours using rlang::inform(). This will keep the author up to date as to which Dev Mode options are being altered. To allow developers a chance to disable Dev Mode messages, the message should be skipped if getOption("shiny.devmode.verbose", TRUE) is not TRUE.

   ```r
   get_devmode_option <- function(name, default = NULL, devmode_default, devmode_message) {
     if (!in_devmode()) {
       # Dev Mode disabled, act like `getOption()`
       return(getOption(name, default = default))
     }

     # Dev Mode enabled, update the default value for `getOption()`
     getOption(name, default = {
       # Notify developer
       if (/
         !missing(devmode_message) &&
         !is.null(devmode_message) &&
         getOption("shiny.devmode.verbose", TRUE)
       ){
         rlang::inform(
           message = devmode_message,
           .frequency = "regularly",
           .frequency_id = devmode_message,
           .file = stderr()
         )
       }
     
     # Notify developer
     if (/
       `getOption()`
     }
   }
   ```
```

The remaining functions in this file are used for author convenience and are not recommended for all reimplemention situations.

Examples

# Enable Shiny Developer mode
devmode()

in_devmode() # TRUE/FALSE?

# Execute code in a temporary shiny dev mode
with_devmode(TRUE, in_devmode()) # TRUE

# Ex: Within shiny, we register the option "shiny.minified"
# to default to 'FALSE' when in Dev Mode
## Not run: register_devmode_option(
  "shiny.minified",
  devmode_default = FALSE
)
## End(Not run)

# Used within `shiny::runApp(launch.browser)`
get_devmode_option("shiny.minified", TRUE) # TRUE if Dev mode is off

is_minified <- with_devmode(TRUE, {
  get_devmode_option("shiny.minified", TRUE)
})

is_minified # FALSE
```

---

<table>
<thead>
<tr>
<th>domains</th>
<th>Reactive domains</th>
</tr>
</thead>
</table>

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.
downloadButton

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,
...,
icon = shiny::icon("download")
)
downloadButton

Arguments

outputId The name of the output slot that the downloadHandler is assigned to.
label The label that should appear on the button.
class Additional CSS classes to apply to the tag, if any.
... Other arguments to pass to the container tag function.
icon An icon() to appear on the button. Default is icon("download").

See Also
downloadHandler()

Examples

```r
## Not run:
ui <- fluidPage(
  p("Choose a dataset to download.
  selectInput("dataset", "Dataset", choices = c("mtcars", "airquality")),
  downloadButton("downloadData", "Download")
)

server <- function(input, output) {
  # The requested dataset
  data <- reactive({
    get(input$dataset)
  })

  output$downloadData <- downloadHandler(
    filename = function() {
      # Use the selected dataset as the suggested file name
      paste0(input$dataset, ".csv")
    },
    content = function(file) {
      # Write the dataset to the `file` that will be downloaded
      write.csv(data(), file)
    }
  )
}

shinyApp(ui, server)

## 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**

```r
downloadHandler(filename, content, contentType = NULL, outputArgs = list())
```

**Arguments**

- `filename`: A string of the filename, including extension, that the user’s web browser should default to when downloading the file; or a function that returns such a string. (Reactive values and functions may be used from this function.)
- `content`: A function that takes a single argument `file` that is a file path (string) of a nonexistent temp file, and writes the content to that file path. (Reactive values and functions may be used from this function.)
- `contentType`: A string of the download’s content type, for example "text/csv" or "image/png". If `NULL`, the content type will be guessed based on the filename extension, or `application/octet-stream` if the extension is unknown.
- `outputArgs`: A list of arguments to be passed through to the implicit call to `downloadButton()` when `downloadHandler` is used in an interactive R Markdown document.

**Examples**

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

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

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

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

Enable bookmarking for a Shiny application

Description

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

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

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

Usage

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

Arguments

store

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

Details

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

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

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

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

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

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

See Also

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

Also see updateQueryString().

Examples

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

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

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

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

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

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

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

enableBookmarking("url")
shinyApp(ui, server)

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

enableBookmarking("url")
shinyApp(ui, server)

# Update the browser's location bar every time an input changes. This should
# not be used with enableBookmarking("server"), because that would create a
# new saved state on disk every time the user changes an input.
ui <- function(req) {
  fluidPage(
    textInput("txt", "Text"),
    checkboxInput("chk", "Checkbox")
  )
}

server <- function(input, output, session) {
  observe(
    # Trigger this observer every time an input changes
    reactiveValuesToList(input)
    session$doBookmark()
  )
  onBookmarked(function(url) {
    updateQueryString(url)
  })
}

enableBookmarking("url")
shinyApp(ui, server)

# Save/restore uploaded files
ui <- function(request) {
  fluidPage(
    sidebarLayout(
      sidebarPanel(
        fileInput("file1", "Choose CSV File", multiple = TRUE,
                    accept = c("text/csv",
                                "text/comma-separated-values,text/plain",
                                ".csv")
                  ),
        tags$hr(),
        checkboxInput("header", "Header", TRUE),
        bookmarkButton()
      ),
      mainPanel(
        tableOutput("contents")
      )
    )
  )
}
exportTestValues

Register expressions for export in test mode

Description

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

Usage

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

Arguments

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

Details

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

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

  options(shiny.testmode = TRUE)

  # This application shows the test snapshot URL; clicking on it will fetch the input, output, and exported values in JSON format.
  shinyApp(
    ui = basicPage(
      h4("Snapshot URL:"),
      uiOutput("url"),
      h4("Current values:"),
      verbatimTextOutput("values"),
      actionButton("inc", "Increment x")
    ),
    server = function(input, output, session) {
      vals <- reactiveValues(x = 1)
      y <- reactive({ vals$x + 1 })

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

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

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

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

---

**ExtendedTask**

**Task or computation that proceeds in the background**

**Description**

In normal Shiny reactive code, whenever an observer, calc, or output is busy computing, it blocks the current session from receiving any inputs or attempting to proceed with any other computation related to that session.
The ExtendedTask class allows you to have an expensive operation that is started by a reactive effect, and whose (eventual) results can be accessed by a regular observer, calc, or output; but during the course of the operation, the current session is completely unblocked, allowing the user to continue using the rest of the app while the operation proceeds in the background.

Note that each ExtendedTask object does not represent a single invocation of its long-running function. Rather, it’s an object that is used to invoke the function with different arguments, keeps track of whether an invocation is in progress, and provides ways to get at the current status or results of the operation. A single ExtendedTask object does not permit overlapping invocations: if the invoke() method is called before the previous invoke() is completed, the new invocation will not begin until the previous invocation has completed.

**ExtendedTask versus asynchronous reactives**

Shiny has long supported using {promises} to write asynchronous observers, calcs, or outputs. You may be wondering what the differences are between those techniques and this class.

Asynchronous observers, calcs, and outputs are not—and have never been—designed to let a user start a long-running operation, while keeping that very same (browser) session responsive to other interactions. Instead, they unblock other sessions, so you can take a long-running operation that would normally bring the entire R process to a halt and limit the blocking to just the session that started the operation. (For more details, see the section on "The Flush Cycle".)

ExtendedTask, on the other hand, invokes an asynchronous function (that is, a function that quickly returns a promise) and allows even that very session to immediately unblock and carry on with other user interactions.

**Methods**

**Public methods:**

- ExtendedTask$new()
- ExtendedTask$invoke()
- ExtendedTask$status()
- ExtendedTask$result()

**Method new():** Creates a new ExtendedTask object. ExtendedTask should generally be created either at the top of a server function, or at the top of a module server function.

*Usage:*

ExtendedTask$new(func)

*Arguments:*

func The long-running operation to execute. This should be an asynchronous function, meaning, it should use the {promises} package, most likely in conjunction with the {future} package. (In short, the return value of func should be a Future object, or a promise, or something else that promises::as.promise() understands.)

It’s also important that this logic does not read from any reactive inputs/sources, as inputs may change after the function is invoked; instead, if the function needs to access reactive inputs, it should take parameters and the caller of the Invoke() method should read reactive inputs and pass them as arguments.
**Method** `invoke()`: Starts executing the long-running operation. If this `ExtendedTask` is already running (meaning, a previous call to `invoke()` is not yet complete) then enqueues this invocation until after the current invocation, and any already-enqueued invocation, completes.

*Usage:*

```plaintext```
ExtendedTask$invoke(...)```
```

*Arguments:*

... Parameters to use for this invocation of the underlying function. If reactive inputs are needed by the underlying function, they should be read by the caller of `invoke` and passed in as arguments.

**Method** `status()`: This is a reactive read that invalidates the caller when the task’s status changes.

*Returns one of the following values:*

- "initial": This `ExtendedTask` has not yet been invoked
- "running": An invocation is currently running
- "success": An invocation completed successfully, and a value can be retrieved via the `result()` method
- "error": An invocation completed with an error, which will be re-thrown if you call the `result()` method

*Usage:*

```plaintext```
ExtendedTask$status()```
```

**Method** `result()`: Attempts to read the results of the most recent invocation. This is a reactive read that invalidates as the task’s status changes.

The actual behavior differs greatly depending on the current status of the task:

- "initial": Throws a silent error (like `req(FALSE)`). If this happens during output rendering, the output will be blanked out.
- "running": Throws a special silent error that, if it happens during output rendering, makes the output appear "in progress" until further notice.
- "success": Returns the return value of the most recent invocation.
- "error": Throws whatever error was thrown by the most recent invocation.

This method is intended to be called fairly naively by any output or reactive expression that cares about the output—you just have to be aware that if the result isn’t ready for whatever reason, processing will stop in much the same way as `req(FALSE)` does, but when the result is ready you’ll get invalidated, and when you run again the result should be there.

Note that the `result()` method is generally not meant to be used with `observeEvent()`, `eventReactive()`, `bindEvent()`, or `isolate()` as the invalidation will be ignored.

*Usage:*

```plaintext```
ExtendedTask$result()```
```

```plaintext```
**fileInput**

*File Upload Control*

**Description**

Create a file upload control that can be used to upload one or more files.

**Usage**

```r
fileInput(
  inputId,
  label,
  multiple = FALSE,
  accept = NULL,
  width = NULL,
  buttonLabel = "Browse...",
  placeholder = "No file selected",
  capture = 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 "unique file type specifiers" which gives the browser a hint as to the type of file the server expects. Many browsers use this prevent the user from selecting an invalid file.
  A unique file type specifier can be:
  - A case insensitive extension like .csv or .rds.
  - A valid MIME type, like text/plain or application/pdf
  - One of audio/*, video/*, or image/* meaning any audio, video, or image type, respectively.
- `width` The width of the input, e.g. '400px', or '100%'; see `validateCssUnit()`.
- `buttonLabel` The label used on the button. Can be text or an HTML tag object.
- `placeholder` The text to show before a file has been uploaded.
- `capture` What source to use for capturing image, audio or video data. This attribute facilitates user access to a device’s media capture mechanism, such as a camera, or microphone, from within a file upload control.
  A value of user indicates that the user-facing camera and/or microphone should be used. A value of environment specifies that the outward-facing camera and/or microphone should be used.
  By default on most phones, this will accept still photos or video. For still photos only, also use accept="image/*". For video only, use accept="video/*".
Details

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

Each time files are uploaded, they are written to a new random subdirectory inside of R’s process-level temporary directory. The Shiny user session keeps track of all uploads in the session, and when the session ends, Shiny deletes all of the subdirectories where files were uploaded to.

Server value

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

- **name**: The filename provided by the web browser. This is **not** the path to read to get at the actual data that was uploaded (see datapath column).
- **size**: The size of the uploaded data, in bytes.
- **type**: The MIME type reported by the browser (for example, text/plain), or empty string if the browser didn’t know.
- **datapath**: The path to a temp file that contains the data that was uploaded. This file may be deleted if the user performs another upload operation.

See Also

Other input elements: `actionButton()`, `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `numericInput()`, `passwordInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, `submitButton()`, `textareaInput()`, `textInput()`, `varSelectInput()`

Examples

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

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      fileInput("file1", "Choose CSV File", accept = ".csv"),
      checkboxInput("header", "Header", TRUE)
    ),
    mainPanel(
      tableOutput("contents")
    )
  )
)

server <- function(input, output) {
  output$contents <- renderTable(
    file <- input$file1
    ext <- tools::file_ext(file$datapath)
    req(file)
    validate(need(ext == "csv", "Please upload a csv file"))}
Create a page that fills the window

Description

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

Usage

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

Arguments

... Elements to include within the page.
padding Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.
title The title to use for the browser window/tab (it will not be shown in the document).
bootstrap If TRUE, load the Bootstrap CSS library.
theme One of the following:
  • NULL (the default), which implies a "stock" build of Bootstrap 3.
  • A bslib::bs_theme() object. This can be used to replace a stock build of Bootstrap 3 with a customized version of Bootstrap 3 or higher.
  • A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css).
fillPage

ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

Details

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

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

For example, fluidPage(plotOutput("plot", height = "100%")) will not work as expected; the plot element’s effective height will be 0, because the plot’s containing elements (<div> and <body>) have automatic height; that is, they determine their own height based on the height of their contained elements. However, fillPage(plotOutput("plot", height = "100%")) will work because fillPage fixes the <body> height at 100% of the window height.

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

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

See Also

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

Examples

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

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

*Flex Box-based row/column layouts*

**Description**

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

**Usage**

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

**Arguments**

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

**Details**

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

To avoid this problem, you have two options:

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

# Only run this example in interactive R sessions.
if (interactive()) {
  ui <- fillPage(fillRow(
    plotOutput("plotLeft", height = "100%"),
    fillCol(
      plotOutput("plotTopRight", height = "100"%),
      plotOutput("plotBottomRight", height = "100%")
    )
  ))
  server <- function(input, output, session) {
    output$plotLeft <- renderPlot(plot(cars))
    output$plotTopRight <- renderPlot(plot(pressure))
    output$plotBottomRight <- renderPlot(plot(AirPassengers))
  }
  shinyApp(ui, server)
}

fixedPage

Create a page with a fixed layout

Description

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

Usage

fixedPage(..., title = NULL, theme = NULL, lang = NULL)

fixedRow(...)
• A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. www/bootstrap.css).

lang
ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

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 definition 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()

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

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

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

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

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

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

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- flowLayout(
    numericInput("rows", "How many rows?", 5),
    selectInput("letter", "Which letter?", LETTERS),
    sliderInput("value", "What value?", 0, 100, 50)
  )
  shinyApp(ui, server = function(input, output) { })
}
```

fluidPage  

Create a page with fluid layout

Description

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

```
fluidPage(..., title = NULL, theme = NULL, lang = 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.

theme  One of the following:

  - NULL (the default), which implies a "stock" build of Bootstrap 3.
  - A `bslib::bs_theme()` object. This can be used to replace a stock build of Bootstrap 3 with a customized version of Bootstrap 3 or higher.
  - A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. `www/bootstrap.css`).

lang  ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the `<html>` tag, as in `<html lang="en">`. The default (NULL) results in an empty string.

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 definition that can be passed to the `shinyUI` function.

Note

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

See Also

`column()`

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

Examples

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

  # Example of UI with fluidPage
  ui <- fluidPage(
```
# Application title
titlePanel("Hello Shiny!")

sidebarLayout(

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

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

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

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

# UI demonstrating column layouts
ui <- fluidPage(
  title = "Hello Shiny!",
  fluidRow(
    column(width = 4, "4"),
    column(width = 3, offset = 2, "3 offset 2")
  )
)

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

---

freezeReactiveVal  
Freeze a reactive value
Description

These functions freeze a `reactiveVal()`, or an element of a `reactiveValues()`. If the value is accessed while frozen, a "silent" exception is raised and the operation is stopped. This is the same thing that happens if `req(FALSE)` is called. The value is thawed (un-frozen; accessing it will no longer raise an exception) when the current reactive domain is flushed. In a Shiny application, this occurs after all of the observers are executed. **NOTE:** We are considering deprecating `freezeReactiveVal`, and `freezeReactiveValue` except when x is `input`. If this affects your app, please let us know by leaving a comment on this GitHub issue.

Usage

```r
freezeReactiveVal(x)

freezeReactiveValue(x, name)
```

Arguments

- **x**
  
  For `freezeReactiveValue`, a `reactiveValues()` object (like `input`); for `freezeReactiveVal`, a `reactiveVal()` object.

- **name**
  
  The name of a value in the `reactiveValues()` object.

See Also

`req()`

Examples

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

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

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

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

Get output information

description

Returns information about the currently executing output, including its name (i.e., outputId); and in some cases, relevant sizing and styling information.

Usage

generalOutputInfo(session = getDefaultReactiveDomain())

Arguments

session

The current Shiny session.

Value

NULL if called outside of an output context; otherwise, a list which includes:

- The name of the output (reported for any output).
- If the output is a plotOutput() or imageOutput(), then:
  - height: a reactive expression which returns the height in pixels.
  - width: a reactive expression which returns the width in pixels.
- If the output is a plotOutput(), imageOutput(), or contains a shiny-report-theme class, then:
  - bg: a reactive expression which returns the background color.
  - fg: a reactive expression which returns the foreground color.
  - accent: a reactive expression which returns the hyperlink color.
  - font: a reactive expression which returns a list of font information, including:
    * families: a character vector containing the CSS font-family property.
    * size: a character string containing the CSS font-size property.
Examples

```r
if (interactive()) {
  shinyApp(
    fluidPage(
      tags$style(HTML("body {background-color: black; color: white; }")),
      tags$style(HTML("body a {color: purple}")),
      tags$style(HTML("#info {background-color: teal; color: orange; }")),
      plotOutput("p"),
      "Computed CSS styles for the output named info:",
      tagAppendAttributes(  # wrap with tagAppendAttributes to ensure HTML is well-formed
        textOutput("info"),
        class = "shiny-report-theme"
      ),
    function(input, output) {
      output$p <- renderPlot({
        info <- getCurrentOutputInfo()
        par(bg = info$bg(), fg = info$fg(), col.axis = info$fg(), col.main = info$fg())
        plot(1:10, col = info$accent(), pch = 19)
        title("A simple R plot that uses its CSS styling")
      })
      output$info <- renderText({
        info <- getCurrentOutputInfo()
        jsonlite::toJSON(
          list(
            bg = info$bg(),
            fg = info$fg(),
            accent = info$accent(),
            font = info$font()
          ),
          auto_unbox = TRUE
        )
      })
    }
  )
}
```

**getQueryString**

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

**Description**

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

**Usage**

```r
getQueryString(session = getDefaultReactiveDomain())
getUrlHash(session = getDefaultReactiveDomain())
```

**Arguments**

- **session** A Shiny session object.

**Details**

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

**Value**

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

**See Also**

- `updateQueryString()`

**Examples**

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

## App 1: getQueryString
## Printing the value of the query string
## (Use the back and forward buttons to see how the browser
## keeps a record of each state)
shinyApp(
  ui = fluidPage(
    textInput("txt", "Enter new query string"),
    helpText("Format: ?param1=val1&param2=val2"),
    actionButton("go", "Update"),
    hr(),
   verbatimTextOutput("query")
  ),
  server = function(input, output, session) {
    observeEvent(input$go, {
      updateQueryString(input$txt, mode = "push")
    })
  })
```
getShinyOption

Get or set Shiny options

Description

There are two mechanisms for working with options for Shiny. One is the options() function, which is part of base R, and the other is the shinyOptions() function, which is in the Shiny package. The reason for these two mechanisms is has to do with legacy code and scoping.

The options() function sets options globally, for the duration of the R process. The getOption() function retrieves the value of an option. All shiny related options of this type are prefixed with "shiny."

The shinyOptions() function sets the value of a shiny option, but unlike options(), it is not always global in scope; the options may be scoped globally, to an application, or to a user session in an application, depending on the context. The getShinyOption() function retrieves a value of a shiny option. Currently, the options set via shinyOptions are for internal use only.
getShinyOption

Usage

getShinyOption(name, default = NULL)

shinyOptions(...)

Arguments

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

Options with options()

shiny.autoreload (defaults to FALSE) If TRUE when a Shiny app is launched, the app directory will be continually monitored for changes to files that have the extensions: r, htm, html, js, css, png, jpg, jpeg, gif. If any changes are detected, all connected Shiny sessions are reloaded. This allows for fast feedback loops when tweaking Shiny UI.
Since monitoring for changes is expensive (we simply poll for last modified times), this feature is intended only for development.
You can customize the file patterns Shiny will monitor by setting the shiny.autoreload.pattern option. For example, to monitor only ui.R: options(shiny.autoreload.pattern = glob2rx("ui.R"))
The default polling interval is 500 milliseconds. You can change this by setting e.g. options(shiny.autoreload.interval = 2000) (every two seconds).

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

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

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

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

shiny.jquery.version (defaults to 3) The major version of jQuery to use. Currently only values of 3 or 1 are supported. If 1, then jQuery 1.12.4 is used. If 3, then jQuery 3.6.0 is used.

shiny.json.digits (defaults to I(16)) Max number of digits to use when converting numbers to JSON format to send to the client web browser. Use I() to specify significant digits. Use NA for max precision.

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

shiny.mathjax.url (defaults to "https://mathjax.rstudio.com/latest/MathJax.js") The URL that should be used to load MathJax, via withMathJax().

shiny.mathjax.config (defaults to "config=TeX-AMS-MML_HTMLorMML") The querystring used to load MathJax, via withMathJax().
shiny.maxRequestSize (defaults to 5MB) This is a number which specifies the maximum web request size, which serves as a size limit for file uploads.

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

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

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

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

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

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

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

shiny.snapshotsortc (defaults to FALSE) If TRUE, test snapshot keys for shinytest will be sorted consistently using the C locale. Snapshots retrieved by shinytest2 will always sort using the C locale.

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

shiny.autoload.r (defaults to TRUE) If TRUE, then the R/ of a shiny app will automatically be sourced.

shiny.useragg (defaults to TRUE) Set to FALSE to prevent PNG rendering via the ragg package. See plotPNG() for more information.

shiny.usecairo (defaults to TRUE) Set to FALSE to prevent PNG rendering via the Cairo package. See plotPNG() for more information.

shiny.devmode (defaults to NULL) Option to enable Shiny Developer Mode. When set, different default getOption(key) values will be returned. See devmode() for more details.

Scoping for shinyOptions()

There are three levels of scoping for shinyOptions(): global, application, and session.

The global option set is available by default. Any calls to shinyOptions() and getShinyOption() outside of an app will access the global option set.
When a Shiny application is run with `runApp()`, the global option set is duplicated and the new option set is available at the application level. If options are set from `global.R`, `app.R`, `ui.R`, or `server.R` (but outside of the server function), then the application-level options will be modified.

Each time a user session is started, the application-level option set is duplicated, for that session. If the options are set from inside the server function, then they will be scoped to the session.

**Options with shinyOptions()**

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

**cache** A caching object that will be used by `renderCachedPlot()`. If not specified, a `cachem::cache_mem()` will be used.

---

**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**

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

```r
htmlOutput(
  outputId,
  inline = FALSE,
  container = if (inline) span else div,
  fill = FALSE,
  ...
)

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

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
- `fill`: If TRUE, the result of `container` is treated as both a fill item and container (see `htmltools::bindFillRole()`), which means both the container as well as its immediate children (i.e., the result of `renderUI()`) are allowed to grow/shrink to fit a fill container with an opinionated height. Set `fill = "item"` or `fill = "container"` to treat container as just a fill item or a fill container.
- ... 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

```r
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, and/or used with `tabPanel()` and `navbarMenu()`.

### Usage

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

#### Arguments

- `name`: The name of the icon. A name from either **Font Awesome** (when `lib="font-awesome"`) or **Bootstrap Glyphicons** (when `lib="glyphicon"`) may be provided. Note that the "fa-" and "glyphicon-" prefixes should not appear in name (i.e., the "fa-calendar" icon should be referred to as "calendar"). A name of `NULL` may also be provided to get a raw `<i>` tag with no library attached to it.

- `class`: Additional classes to customize the style of an icon (see the usage examples for details on supported styles).

- `lib`: The icon library to use. Either "font-awesome" or "glyphicon".

- `...`: Arguments passed to the `<i>` tag of `htmltools::tags`.

### Value

An `<i>` (icon) HTML tag.

### See Also

For lists of available icons, see [https://fontawesome.com/icons](https://fontawesome.com/icons) and [https://getbootstrap.com/docs/3.3/components/#glyphicons](https://getbootstrap.com/docs/3.3/components/#glyphicons)
Examples

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

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

inputPanel

Description

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

Usage

inputPanel(...)

Arguments

... Input controls or other HTML elements.

insertTab

Description

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

Usage

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

prependTab(
    inputId,
insertTab

```r
insertTab(
  tab,
  select = FALSE,
  menuName = NULL,
  session = getDefaultReactiveDomain()
)
```

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

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

**Arguments**

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

**Details**

When you want to insert a new tab before or after an existing tab, you should use `insertTab`. When you want to prepend a tab (i.e. add a tab to the beginning of the tabsetPanel), use `prependTab`. When you want to append a tab (i.e. add a tab to the end of the tabsetPanel), use `appendTab`. For navbarPage, you can insert/remove conventional tabPanels (whether at the top level or nested inside a navbarMenu), as well as an entire `navbarMenu()`. For the latter case, `target` should be the `menuName` that you gave your `navbarMenu` when you first created it (by default, this is equal to the value of the `title` argument).

**See Also**

- `showTab()`
Examples

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

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

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

shinyApp(ui, server)

# example app for prepending/appending a navbarMenu
ui <- navbarPage("Navbar page", id = "tabs",
  tabPanel("Home",
    actionButton("prepend", "Prepend a navbarMenu"),
    actionButton("append", "Append a navbarMenu")
  )
)

server <- function(input, output, session) {
  observeEvent(input$prepend, {
    id <- paste0("Dropdown", input$prepend, "p")
    prependTab(inputId = "tabs",
      navMenu(id,
        tabPanel("Drop1", paste("Drop1 page from", id)),
        tabPanel("Drop2", paste("Drop2 page from", id)),
        "------",
        "Header",
        tabPanel("Drop3", paste("Drop3 page from", id)))
  })
  observeEvent(input$append, {
    appendTab(inputId = "tabs",
      navMenu(id,
        tabPanel("Drop1", paste("Drop1 page from", id)),
        tabPanel("Drop2", paste("Drop2 page from", id)),
        "------",
        "Header",
        tabPanel("Drop3", paste("Drop3 page from", id)))
  })
}

shinyApp(ui, server)


```r
observeEvent(input$append, {
  id <- paste0("Dropdown", input$append, "a")
  appendTab(inputId = "tabs",
    navbarMenu(id,
      tabPanel("Drop1", paste("Drop1 page from", id)),
      tabPanel("Drop2", paste("Drop2 page from", id)),
      "------",
      "Header",
      tabPanel("Drop3", paste("Drop3 page from", id))
    )
  )
})

shinyApp(ui, server)
```

---

### insertUI

**Insert and remove UI objects**

#### Description

These functions allow you to dynamically add and remove arbitrary UI into your app, whenever you want, as many times as you want. Unlike `renderUI()`, the UI generated with `insertUI()` is persistent: once it’s created, it stays there until removed by `removeUI()`. Each new call to `insertUI()` creates more UI objects, in addition to the ones already there (all independent from one another). To update a part of the UI (ex: an input object), you must use the appropriate `render` function or a customized `reactive` function.

#### Usage

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

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

Arguments

selector  A string that is accepted by jQuery’s selector (i.e. the string s to be placed in a $(s) jQuery call).

For insertUI() this determines the element(s) relative to which you want to insert your UI object. For removeUI() this determine the element(s) to be removed. If you want to remove a Shiny input or output, note that many of these are wrapped in <div>s, so you may need to use a somewhat complex selector — see the Examples below. (Alternatively, you could also wrap the inputs/outputs that you want to be able to remove easily in a <div> with an id.)

where  Where your UI object should go relative to the selector:

beforeBegin  Before the selector element itself
afterBegin  Just inside the selector element, before its first child
beforeEnd  Just inside the selector element, after its last child (default)
afterEnd  After the selector element itself

Adapted from https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML.

ui  The UI object you want to insert. This can be anything that you usually put inside your app’s ui function. If you’re inserting multiple elements in one call, make sure to wrap them in either a tagList() or a tags$div() (the latter option has the advantage that you can give it an id to make it easier to reference or remove it later on). If you want to insert raw html, use ui = HTML().

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

immediate  Whether the UI object should be immediately inserted or removed, or whether Shiny should wait until all outputs have been updated and all observers have been run (default).

session  The shiny session. Advanced use only.

Details

It’s particularly useful to pair removeUI with insertUI(), but there is no restriction on what you can use it on. Any element that can be selected through a jQuery selector can be removed through this function.

Examples

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

  # Server logic
  server <- function(input, output, session) {

    # Insert UI
    output$ui <- insertUI(selector = ui, where = "afterBegin", ui = ui, immediate = TRUE)
invalidateLater

Scheduled Invalidation

Description

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

Usage

invalidateLater(millis, session = getDefaultReactiveDomain())

Arguments

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

Details

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

See Also

reactiveTimer() is a slightly less safe alternative.

Examples

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

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

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

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

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

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

  }

  shinyApp(ui, server)
}
```
is.reactivevalues  Checks whether an object is a reactivevalues object

Description
Checks whether its argument is a reactivevalues object.

Usage
is.reactivevalues(x)

Arguments
x The object to test.

See Also
reactiveValues().

isolate  Create a non-reactive scope for an expression

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

Usage
isolate(expr)

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

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

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

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

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

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

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

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

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

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

## End(Not run)

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

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

---

`isRunning`  
Check whether a Shiny application is running
Description

This function tests whether a Shiny application is currently running.

Usage

isRunning()

Value

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

---

isTruthy

Truthy and falsy values

Description

The terms "truthy" and "falsy" generally indicate whether a value, when coerced to a `base::logical()`, is TRUE or FALSE. We use the term a little loosely here; our usage tries to match the intuitive notions of "Is this value missing or available?", or "Has the user provided an answer?", or in the case of action buttons, "Has the button been clicked?".

Usage

isTruthy(x)

Arguments

x  An expression whose truthiness value we want to determine

Details

For example, a `textInput` that has not been filled out by the user has a value of "", so that is considered a falsy value.

To be precise, a value is truthy unless it is one of:

- FALSE
- NULL
- ""
- An empty atomic vector
- An atomic vector that contains only missing values
- A logical vector that contains all FALSE or missing values
- An object of class "try-error"
- A value that represents an unclicked `actionButton()`

Note in particular that the value 0 is considered truthy, even though `as.logical(0)` is FALSE.
loadSupport

Load an app’s supporting R files

Description


Usage

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

Arguments

appDir The application directory. If appDir is NULL or not supplied, the nearest enclosing directory that is a Shiny app, starting with the current directory, is used.
renv The environment in which the files in the R/ directory should be evaluated.
globalrenv The environment in which global.R should be evaluated. If NULL, global.R will not be evaluated at all.

Details

Since Shiny 1.5.0, this function is called by default when running an application. If it causes problems, there are two ways to opt out. You can either place a file named _disable_autoload.R in your R/ directory, or set options(shiny.autoload.r=FALSE). If you set this option, it will affect any application that runs later in the same R session, potentially breaking it, so after running your application, you should unset option with options(shiny.autoload.r=NULL)
The files are sourced in alphabetical order (as determined by list.files). global.R is evaluated before the supporting R files in the R/ directory.

markdown

Insert inline Markdown

Description

This function accepts Markdown-syntax text and returns HTML that may be included in Shiny UIs.

Usage

markdown(mds, extensions = TRUE, .noWS = NULL, ...)

markdown

Insert inline Markdown

Description

This function accepts Markdown-syntax text and returns HTML that may be included in Shiny UIs.

Usage

markdown(mds, extensions = TRUE, .noWS = NULL, ...)

markdown

Insert inline Markdown

Description

This function accepts Markdown-syntax text and returns HTML that may be included in Shiny UIs.

Usage

markdown(mds, extensions = TRUE, .noWS = NULL, ...)

markdown

Insert inline Markdown

Description

This function accepts Markdown-syntax text and returns HTML that may be included in Shiny UIs.

Usage

markdown(mds, extensions = TRUE, .noWS = NULL, ...)

markdown

Insert inline Markdown

Description

This function accepts Markdown-syntax text and returns HTML that may be included in Shiny UIs.

Usage

markdown(mds, extensions = TRUE, .noWS = NULL, ...)
markdown

Arguments

- **mds**: A character vector of Markdown source to convert to HTML. If the vector has more than one element, a single-element character vector of concatenated HTML is returned.
- **extensions**: Enable Github syntax extensions; defaults to `TRUE`.
- **.noWS**: Character vector used to omit some of the whitespace that would normally be written around generated HTML. Valid options include `before`, `after`, and `outside` (equivalent to `before` and `end`).
- **...**: Additional arguments to pass to `commonmark::markdown_html()`. These arguments are *dynamic*.

Details

Leading whitespace is trimmed from Markdown text with `glue::trim()`. Whitespace trimming ensures Markdown is processed correctly even when the call to `markdown()` is indented within surrounding R code.

By default, Github extensions are enabled, but this can be disabled by passing `extensions = FALSE`. Markdown rendering is performed by `commonmark::markdown_html()`. Additional arguments to `markdown()` are passed as arguments to `markdown_html()`.

Value

A character vector marked as HTML.

Examples

```r
ui <- fluidPage(
  markdown("#
    This is a markdown paragraph, and will be contained within a `<p>` tag in the UI.

    The following is an unordered list, which will be represented in the UI as a `<ul>` with `<li>` children:

    * a bullet
    * another

    [Links](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a) work; so does *emphasis*.

    To see more of what's possible, check out [commonmark.org/help](https://commonmark.org/help).
  
")
)
```
MarkRenderFunction  Mark a function as a render function

Description

[Superseded] Please use createRenderFunction() to support async execution. (Shiny 1.1.0)

Usage

markRenderFunction(
  uiFunc,
  renderFunc,
  outputArgs = list(),
  cacheHint = "auto",
  cacheWriteHook = NULL,
  cacheReadHook = NULL
)

Arguments

uiFunc  A function that renders Shiny UI. Must take a single argument: an output ID.
renderFunc  A function that is suitable for assigning to a Shiny output slot.
outputArgs  A list of arguments to pass to the uiFunc. Render functions should include outputArgs = list() in their own parameter list, and pass through the value to markRenderFunction, to allow app authors to customize outputs. (Currently, this is only supported for dynamically generated UIs, such as those created by Shiny code snippets embedded in R Markdown documents).
cacheHint  One of "auto", FALSE, or some other information to identify this instance for caching using bindCache(). If "auto", it will try to automatically infer caching information. If FALSE, do not allow caching for the object. Some render functions (such as renderPlot) contain internal state that makes them unsuitable for caching.
cacheWriteHook  Used if the render function is passed to bindCache(). This is an optional callback function to invoke before saving the value from the render function to the cache. This function must accept one argument, the value returned from renderFunc, and should return the value to store in the cache.
cacheReadHook  Used if the render function is passed to bindCache(). This is an optional callback function to invoke after reading a value from the cache (if there is a cache hit). The function will be passed one argument, the value retrieved from the cache. This can be useful when some side effect needs to occur for a render function to behave correctly. For example, some render functions call createWebDependency() so that Shiny is able to serve JS and CSS resources.
**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()`
MockShinySession

Description

An R6 class suitable for testing purposes. Simulates, to the extent possible, the behavior of the ShinySession class. The session parameter provided to Shiny server functions and modules is an instance of a ShinySession in normal operation.

Most kinds of module and server testing do not require this class be instantiated manually. See instead testServer().

In order to support advanced usage, instances of MockShinySession are unlocked so that public methods and fields of instances may be modified. For example, in order to test authentication workflows, the user or groups fields may be overridden. Modified instances of MockShinySession may then be passed explicitly as the session argument of testServer().

Public fields

eenv The environment associated with the session.
  returned The value returned by the module under test.
  singletons Hardcoded as empty. Needed for rendering HTML (i.e. renderUI).
  clientData Mock client data that always returns a size for plots.
  output The shinyoutputs associated with the session.
  input The reactive inputs associated with the session.
  userData An environment initialized as empty.
  progressStack A stack of progress objects.
  token On a real ShinySession, used to identify this instance in URLs.
  cache The session cache object.
  appcache The app cache object.
  restoreContext Part of bookmarking support in a real ShinySession but always NULL for a MockShinySession.
  groups Character vector of groups associated with an authenticated user. Always NULL for a MockShinySession.
  user The username of an authenticated user. Always NULL for a MockShinySession.
  options A list containing session-level shinyOptions.

Active bindings

  files For internal use only.
  downloads For internal use only.
  closed Deprecated in ShinySession and signals an error.
  session Deprecated in ShinySession and signals an error.
  request An empty environment where the request should be. The request isn’t meaningfully mocked currently.
**MockShinySession**

**Methods**

**Public methods:**

- `MockShinySession$new()`  
- `MockShinySession$onFlush()`  
- `MockShinySession$onFlushed()`  
- `MockShinySession$onEnded()`  
- `MockShinySession$isEnded()`  
- `MockShinySession$isClosed()`  
- `MockShinySession$close()`  
- `MockShinySession$cycleStartAction()`  
- `MockShinySession$fileUrl()`  
- `MockShinySession$setInputs()`  
- `MockShinySession$scheduleTask()`  
- `MockShinySession$elapse()`  
- `MockShinySession$.now()`  
- `MockShinySession$defineOutput()`  
- `MockShinySession$getOutput()`  
- `MockShinySession$ns()`  
- `MockShinySession$flushReact()`  
- `MockShinySession$makeScope()`  
- `MockShinySession$setEnv()`  
- `MockShinySession$setReturned()`  
- `MockShinySession$getReturned()`  
- `MockShinySession$genId()`  
- `MockShinySession$rootScope()`  
- `MockShinySession$onUnhandledError()`  
- `MockShinySession$unhandledError()`  
- `MockShinySession$freezeValue()`  
- `MockShinySession$onSessionEnded()`  
- `MockShinySession$registerDownload()`  
- `MockShinySession$getCurrentOutputInfo()`  
- `MockShinySession$clone()`

**Method** `new()`  
Create a new MockShinySession.

*Usage:*

`MockShinySession$new()`

**Method** `onFlush()`  
Define a callback to be invoked before a reactive flush

*Usage:*

`MockShinySession$onFlush(fun, once = TRUE)`

*Arguments:*

- `fun` The function to invoke
once  If TRUE, will only run once. Otherwise, will run every time reactives are flushed.

**Method onFlushed():** Define a callback to be invoked after a reactive flush

*Usage:*
MockShinySession$onFlushed(fun, once = TRUE)

*Arguments:*

fun  The function to invoke

once  If TRUE, will only run once. Otherwise, will run every time reactives are flushed.

**Method onEnded():** Define a callback to be invoked when the session ends

*Usage:*
MockShinySession$onEnded(sessionEndedCallback)

*Arguments:*

sessionEndedCallback  The callback to invoke when the session has ended.

**Method isEnded():** Returns FALSE if the session has not yet been closed

*Usage:*
MockShinySession$isEnded()

**Method isOpen():** Returns FALSE if the session has not yet been closed

*Usage:*
MockShinySession$isClosed()

**Method close():** Closes the session

*Usage:*
MockShinySession$close()

**Method cycleStartAction():** Unsophisticated mock implementation that merely invokes

*Usage:*
MockShinySession$cycleStartAction(callback)

*Arguments:*

callback  The callback to be invoked.

**Method fileUrl():** Base64-encode the given file. Needed for image rendering.

*Usage:*
MockShinySession$fileUrl(name, file, contentType = "application/octet-stream")

*Arguments:*

name  Not used

file  The file to be encoded

contentType  The content type of the base64-encoded string

**Method setInputs():** Sets reactive values associated with the session$inputs object and flushes the reactives.

*Usage:*

MockShinySession$setInputs()
MockShinySession$setInputs(...)

Arguments:
... The inputs to set. These arguments are processed with `rlang::list2()` and so are dynamic. Input names may not be duplicated.

Examples:
```r
dontrun{
  session$setInputs(x=1, y=2)
}
```

**Method .scheduleTask():** An internal method which shouldn’t be used by others. Schedules callback for execution after some number of millis milliseconds.

Usage:
```r
MockShinySession$.scheduleTask(millis, callback)
```

Arguments:
- `millis` The number of milliseconds on which to schedule a callback
- `callback` The function to schedule.

**Method .elapse():** Simulate the passing of time by the given number of milliseconds.

Usage:
```r
MockShinySession$.elapse(millis)
```

Arguments:
- `millis` The number of milliseconds to advance time.

**Method .now():** An internal method which shouldn’t be used by others.

Usage:
```r
MockShinySession$.now()
```

Returns: Elapsed time in milliseconds.

**Method $defineOutput():** An internal method which shouldn’t be used by others. Defines an output in a way that sets private$currentOutputName appropriately.

Usage:
```r
MockShinySession$defineOutput(name, func, label)
```

Arguments:
- `name` The name of the output.
- `func` The render definition.
- `label` Not used.

**Method $getOutput():** An internal method which shouldn’t be used by others. Forces evaluation of any reactive dependencies of the output function.

Usage:
```r
MockShinySession$getOutput(name)
```

Arguments:
- `name` The name of the output.
Returns: The return value of the function responsible for rendering the output.

Method `ns()`: Returns the given id prefixed by this namespace’s id.

Usage:
MockShinySession$ns(id)

Arguments:
id The id to prefix with a namespace id.

Returns: The id with a namespace prefix.

Method `flushReact()`: Trigger a reactive flush right now.

Usage:
MockShinySession$flushReact()

Method `makeScope()`: Create and return a namespace-specific session proxy.

Usage:
MockShinySession$makeScope(namespace)

Arguments:
namespace Character vector indicating a namespace.

Returns: A new session proxy.

Method `setEnv()`: Set the environment associated with a testServer() call, but only if it has not previously been set. This ensures that only the environment of the outermost module under test is the one retained. In other words, the first assignment wins.

Usage:
MockShinySession$setEnv(env)

Arguments:
env The environment to retain.

Returns: The provided env.

Method `setReturned()`: Set the value returned by the module call and proactively flush. Note that this method may be called multiple times if modules are nested. The last assignment, corresponding to an invocation of `setReturned()` in the outermost module, wins.

Usage:
MockShinySession$setReturned(value)

Arguments:
value The value returned from the module

Returns: The provided value.

Method `getReturned()`: Get the value returned by the module call.

Usage:
MockShinySession$getReturned()

Returns: The value returned by the module call
Method `genId()`: Generate a distinct character identifier for use as a proxy namespace.

Usage:
MockShinySession$genId()

Returns: A character identifier unique to the current session.

Method `rootScope()`: Provides a way to access the root `MockShinySession` from any descendant proxy.

Usage:
MockShinySession$rootScope()

Returns: The root `MockShinySession`.

Method `onUnhandledError()`: Add an unhandled error callback.

Usage:
MockShinySession$onUnhandledError(callback)

Arguments:
- callback The callback to add, which should accept an error object as its first argument.

Returns: A deregistration function.

Method `unhandledError()`: Called by observers when a reactive expression errors.

Usage:
MockShinySession$unhandledError(e, close = TRUE)

Arguments:
- e An error object.
- close If TRUE, the session will be closed after the error is handled, defaults to FALSE.

Method `freezeValue()`: Freeze a value until the flush cycle completes.

Usage:
MockShinySession$freezeValue(x, name)

Arguments:
- x A `ReactiveValues` object.
- name The name of a reactive value within x.

Method `onSessionEnded()`: Registers the given callback to be invoked when the session is closed (i.e. the connection to the client has been severed). The return value is a function which unregisters the callback. If multiple callbacks are registered, the order in which they are invoked is not guaranteed.

Usage:
MockShinySession$onSessionEnded(sessionEndedCallback)

Arguments:
- sessionEndedCallback Function to call when the session ends.

Method `registerDownload()`: Associated a downloadable file with the session.

Usage:
MockShinySession$registerDownload(name, filename, contentType, content)

**Arguments:**
- `name`  The un-namespaced output name to associate with the downloadable file.
- `filename`  A string or function designating the name of the file.
- `contentType`  A string of the content type of the file. Not used by MockShinySession.
- `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.)

**Method** `getCurrentOutputInfo()`: Get information about the output that is currently being executed.

**Usage:**
MockShinySession$getCurrentOutputInfo()

**Returns:** A list with the `name` of the output. If no output is currently being executed, this will return `NULL` output, or `NULL` if no output is currently executing.

**Method** `clone()`: The objects of this class are cloneable with this method.

**Usage:**
MockShinySession$clone(deep = FALSE)

**Arguments:**
- `deep`  Whether to make a deep clone.

### Examples
```r
## ------------------------------------------------
## Method `MockShinySession$setInputs`
## ------------------------------------------------
## Not run:
session$setInputs(x=1, y=2)
## End(Not run)
```

---

**modalDialog**

Create a modal dialog UI

### Description

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

`modalButton()` creates a button that will dismiss the dialog when clicked, typically used when customising the footer.
Usage

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

modalButton(label, icon = NULL)

Arguments

...  UI elements for the body of the modal dialog box.
title  An optional title for the dialog.
footer  UI for footer. Use NULL for no footer.
size  One of "s" for small, "m" (the default) for medium, "l" for large, or "xl" for extra large. Note that "xl" only works with Bootstrap 4 and above (to opt-in to Bootstrap 4+, pass `bslib::bs_theme()` to the theme argument of a page container like `fluidPage()`).
easyClose  If TRUE, the modal dialog can be dismissed by clicking outside the dialog box, or be pressing the Escape key. If FALSE (the default), the modal dialog can’t be dismissed in those ways; instead it must be dismissed by clicking on a modalButton(), or from a call to `removeModal()` on the server.
fade  If FALSE, the modal dialog will have no fade-in animation (it will simply appear rather than fade in to view).
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.

Examples

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

# Display a modal that requires valid input before continuing.
shinyApp(
  ui = basicPage(
    actionButton("show", "Show modal dialog"),
    verbatimTextOutput("dataInfo")
  ),
  server = function(input, output) {
    # reactiveValues object for storing current data set.
    vals <- reactiveValues(data = NULL)

    # Return the UI for a modal dialog with data selection input. If 'failed' is
    # TRUE, then display a message that the previous value was invalid.
    dataModal <- function(failed = FALSE) {
      modalDialog(
        input = "dataset", "Choose data set",
        placeholder = 'Try "mtcars" or "abc"'
      ),
      span('Try the name of a valid data object like "mtcars", "
        then a name of a non-existent object like "abc"
      ')
    }

    server = reactiveValues(data = NULL)

    # Return the UI for a modal dialog with data selection input. If 'failed' is
    # TRUE, then display a message that the previous value was invalid.
    dataModal <- function(failed = FALSE) {
      modalDialog(
        textInput("dataset", "Choose data set",
          placeholder = 'Try "mtcars" or "abc"'
        ),
        span('Try the name of a valid data object like "mtcars", "
          then a name of a non-existent object like "abc"
        ')
      )
    }
    
    if (failed) {
      div(tags$b("Invalid name of data object", style = "color: red;")),
      footer = tagList(
        modalButton("Cancel"),
        actionButton("ok", "OK")
      )
    }
  })
)
# Show modal when button is clicked.
observeEvent(input$show, {
    showModal(dataModal())
});

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

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

Details

Starting in Shiny 1.5.0, we recommend using `moduleServer` instead of `callModule()`, because the syntax is a little easier to understand, and modules created with `moduleServer` can be tested with `testServer()`.

Value

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

See Also

https://shiny.rstudio.com/articles/modules.html

Examples

```r
# Define the UI for a module
counterUI <- function(id, label = "Counter") {
  ns <- NS(id)
  tagList(
    actionButton(ns("button"), label = label),
    verbatimTextOutput(ns("out"))
  )
}

# Define the server logic for a module
counterServer <- function(id) {
  moduleServer(
    id,
    function(input, output, session) {
      count <- reactiveVal(0)
      observeEvent(input$button, {
        count(count() + 1)
      })
      output$out <- renderText({
        count()
      })
      count
    }
  )
}

# Use the module in an app
ui <- fluidPage(
  counterUI("counter1", "Counter #1"),
  counterUI("counter2", "Counter #2")
)
server <- function(input, output, session) {
  counterServer("counter1")
  counterServer("counter2")
}
if (interactive()) {
  shinyApp(ui, server)
```
# If you want to pass extra parameters to the module’s server logic, you can # add them to your function. In this case „prefix“ is text that will be # printed before the count.
counterServer2 <- function(id, prefix = NULL) {
  moduleServer(
    id,
    function(input, output, session) {
      count <- reactiveVal(0)
      observeEvent(input$button, {
        count(count() + 1)
      })
      output$out <- renderText({
        paste0(prefix, count())
      })
      count
    }
  )
}

ui <- fluidPage(
  counterUI("counter", "Counter"),
)
server <- function(input, output, session) {
  counterServer2("counter", "The current count is: ")
}
if (interactive()) {
  shinyApp(ui, server)
}

navbarPage

\[ \textbf{navbarPage} \]

Create a page with a top level navigation bar

\[ \textbf{Description} \]

Create a page that contains a top level navigation bar that can be used to toggle a set of \texttt{tabPanel()} elements.

\[ \textbf{Usage} \]

\[
\text{navbarPage}( \text{title, ...}, \text{id = NULL, selected = NULL, position outbreak = c("static-top", "fixed-top", "fixed-bottom")},
\]
navbarPage

```r
header = NULL,
footer = NULL,
inverse = FALSE,
collapsible = FALSE,
fluid = TRUE,
theme = NULL,
windowTitle = NA,
lang = NULL
)

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

**Arguments**

- **title**: The title to display in the navbar.
- **...**: `tabPanel()` elements to include in the page. The `navbarMenu` function also accepts strings, which will be used as menu section headers. If the string is a set of dashes like "-----" a horizontal separator will be displayed in the menu.
- **id**: If provided, you can use `input$id` in your server logic to determine which of the current tabs is active. The value will correspond to the value argument that is passed to `tabPanel()`.
- **selected**: The value (or, if none was supplied, the title) of the tab that should be selected by default. If NULL, the first tab will be selected.
- **position**: Determines whether the navbar should be displayed at the top of the page with normal scrolling behavior ("static-top"), pinned at the top ("fixed-top"), or pinned at the bottom ("fixed-bottom"). Note that using "fixed-top" or "fixed-bottom" will cause the navbar to overlay your body content, unless you add padding, e.g.: `tags$style(type="text/css", "body {padding-top: 70px;}")`
- **header**: Tag or list of tags to display as a common header above all tabPanels.
- **footer**: Tag or list of tags to display as a common footer below all tabPanels.
- **inverse**: TRUE to use a dark background and light text for the navigation bar.
- **collapsible**: TRUE to automatically collapse the navigation elements into an expandable menu on mobile devices or narrow window widths.
- **fluid**: TRUE to use a fluid layout. FALSE to use a fixed layout.
- **theme**: One of the following:
  - NULL (the default), which implies a "stock" build of Bootstrap 3.
  - A `bslib::bs_theme()` object. This can be used to replace a stock build of Bootstrap 3 with a customized version of Bootstrap 3 or higher.
  - A character string pointing to an alternative Bootstrap stylesheet (normally a css file within the www directory, e.g. `www/bootstrap.css`).
- **windowTitle**: the browser window title (as a character string). The default value, NA, means to use any character strings that appear in title (if none are found, the host URL of the page is displayed by default).
**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.

**Details**

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.

**Value**

A UI definition that can be passed to the `shinyUI` function.

**See Also**

`tabPanel()`, `tabsetPanel()`, `updateNavbarPage()`, `insertTab()`, `showTab()`

Other layout functions: `fillPage()`, `fixedPage()`, `flowLayout()`, `fluidPage()`, `sidebarLayout()`, `splitLayout()`, `verticalLayout()`

**Examples**

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

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

**navlistPanel**

ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the `<html>` tag, as in `<html lang="en">`. The default (NULL) results in an empty string.

**menuName**

A name that identifies this `navbarMenu`. This is needed if you want to insert/remove or show/hide an entire `navbarMenu`.

**icon**

Optional icon to appear on a `navbarMenu` tab.
Usage

```r
navlistPanel(
  ..., 
  id = NULL, 
  selected = NULL, 
  header = NULL, 
  footer = 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.
- `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.
- `well` `TRUE` to place a well (gray rounded rectangle) around the navigation list. `FALSE` to use fixed layout.
- `fluid` `TRUE` to use fluid layout; `FALSE` to use fixed layout.
- `widths` Column widths of the navigation list and tabset content areas respectively.

Details

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

See Also

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

Examples

```r
fluidPage(
  titlePanel("Application Title"),
  navlistPanel("Header", 
    tabPanel("First"),
    tabPanel("Second"),
  )
)
```
Namespaced IDs for inputs/outputs

Description

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

Usage

NS(namespace, id = NULL)

ns.sep

Arguments

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

id The id string to be namespaced (optional).

Format

An object of class character of length 1.

Details

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

Value

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

See Also

https://shiny.rstudio.com/articles/modules.html
numericInput

Create a numeric input control

Description

Create an input control for entry of numeric values

Usage

numericInput(
  inputId,  # 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 = NA, # Minimum allowed value
  max = NA, # Maximum allowed value
  step = NA, # Interval to use when stepping between min and max
  width = NULL
)

Arguments

inputId
  Display label for the control, or NULL for no label.
value
  Initial value.
min
  Minimum allowed value
max
  Maximum allowed value
step
  Interval to use when stepping between min and max
width
  The width of the input, e.g. ‘400px’, or ‘100%’; see validateCssUnit().

Value

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

Server value

A numeric vector of length 1.

See Also

updateNumericInput()

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

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

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

observe  Create a reactive observer

Description

Creates an observer from the given expression.

Usage

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

Arguments

x  An expression (quoted or unquoted). Any return value will be ignored.

env  The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If x is a quosure and quoted is TRUE, then env is ignored.

quoted  If it is TRUE, then the quote(ED value of x will be used when x is evaluated. If x is a quosure and you would like to use its expression as a value for x, then you must set quoted to TRUE.

...  Not used.
label  A label for the observer, useful for debugging.
suspended  If TRUE, start the observer in a suspended state. If FALSE (the default), start in a non-suspended state.
priority  An integer or numeric that controls the priority with which this observer should be executed. A higher value means higher priority: an observer with a higher priority value will execute before all observers with lower priority values. Positive, negative, and zero values are allowed.
domain  See domains.
autoDestroy  If TRUE (the default), the observer will be automatically destroyed when its domain (if any) ends.

.. stacktraceon  Advanced use only. For stack manipulation purposes; see stacktrace().

Details

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

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

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

Value

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

suspend()  Causes this observer to stop scheduling flushes (re-executions) in response to invalidations. If the observer was invalidated prior to this call but it has not re-executed yet then that re-execution will still occur, because the flush is already scheduled.
resume()  Causes this observer to start re-executing in response to invalidations. If the observer was invalidated while suspended, then it will schedule itself for re-execution.
destroy()  Stops the observer from executing ever again, even if it is currently scheduled for re-execution.
setPriority(priority = 0)  Change this observer’s priority. Note that if the observer is currently invalidated, then the change in priority will not take effect until the next invalidation–unless the observer is also currently suspended, in which case the priority change will be effective upon resume.
setAutoDestroy(autoDestroy)  Sets whether this observer should be automatically destroyed when its domain (if any) ends. If autoDestroy is TRUE and the domain already ended, then destroy() is called immediately.

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

values <- reactiveValues(A=1)

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

# To store expressions for later conversion to observe, use rlang::quo()
myquo <- rlang::quo({print(values$A + 3)})
obsC <- rlang::inject(observe(!myquo))

# (Legacy) Can use quoted expressions
obsD <- observe(quote({print(values$A + 2)}), 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. As of Shiny 1.6.0, we recommend using bindEvent() instead of eventReactive() and observeEvent(). This is because bindEvent() can be composed with bindCache(), and because it can also be used with render functions (like renderText() and renderPlot()).

Usage

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

Arguments

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

**handlerExpr**
The expression to call whenever `eventExpr` is invalidated. This should be a side-effect-producing action (the return value will be ignored). It will be executed within an `isolate()` scope.

**event.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. If `eventExpr` is a quosure and `event.quoted` is `TRUE`, then `event.env` is ignored.

**event.quoted**
If it is `TRUE`, then the `quote()`d value of `eventExpr` will be used when `eventExpr` is evaluated. If `eventExpr` is a quosure and you would like to use its expression as a value for `eventExpr`, then you must set `event.quoted` to `TRUE`.

**handler.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. If `handlerExpr` is a quosure and `handler.quoted` is `TRUE`, then `handler.env` is ignored.

**handler.quoted**
If it is `TRUE`, then the `quote()`d value of `handlerExpr` will be used when `handlerExpr` is evaluated. If `handlerExpr` is a quosure and you would like to use its expression as a value for `handlerExpr`, then you must set `handler.quoted` to `TRUE`.

**...**
Currently not used.

**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 ends.

**ignoreNULL**  Whether the action should be triggered (or value calculated, in the case of eventReactive) when the input event expression is NULL. See Details.

**ignoreInit**  If TRUE, then, when this observeEvent is first created/initialized, ignore the handlerExpr (the second argument), whether it is otherwise supposed to run or not. The default is FALSE. See Details.

**once**  Whether this observeEvent should be immediately destroyed after the first time that the code in handlerExpr is run. This pattern is useful when you want to subscribe to an event that should only happen once.

**valueExpr**  The expression that produces the return value of the eventReactive. It will be executed within an `isolate()` scope.

**value.env**  The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If `valueExpr` is a quosure and `value.quoted` is TRUE, then `value.env` is ignored.

**value.quoted**  If it is TRUE, then the `quote()`d value of `valueExpr` will be used when `valueExpr` is evaluated. If `valueExpr` is a quosure and you would like to use its expression as a value for `valueExpr`, then you must set `value.quoted` to TRUE.

**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. Note that `observeEvent()` is equivalent to using `observe()` %>% `bindEvent()` and as of Shiny 1.6.0, we recommend the latter.

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. Note that `eventReactive()` is equivalent to using `reactive()` %>% `bindEvent()` and as of Shiny 1.6.0, we recommend the latter.

**Value**

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

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

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

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

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

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

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

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

See Also

actionButton()

Examples

## Only run examples in interactive R sessions
if (interactive()) {
  ## App 1: Sample usage
  shinyApp(
    ui = fluidPage(
      column(4,
observeEvent

```r
numericInput("x", "Value", 5),
br(),
actionButton("button", "Show")
),
column(8, tableOutput("table"))
),
server = function(input, output) {
  # Take an action every time button is pressed;
  # here, we just print a message to the console
  observeEvent(input$button, {
    cat("Showing", input$x, "\n")
  })
  # The observeEvent() above is equivalent to:
  # observe({
  #   cat("Showing", input$x, "\n")
  # }) %>%
  # bindEvent(input$button)

  # Take a reactive dependency on input$button, but
  # not on any of the stuff inside the function
  df <- eventReactive(input$button, {
    head(cars, input$x)
  })
  output$table <- renderTable(df)
}
```

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

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

Add callbacks for Shiny session bookmarking events

Description

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

- `onBookmark` registers a function that will be called just before Shiny bookmarks state.
- `onBookmarked` registers a function that will be called just after Shiny bookmarks state.
- `onRestore` registers a function that will be called when a session is restored, after the server function executes, but before all other reactives, observers and render functions are run.
- `onRestored` registers a function that will be called after a session is restored. This is similar to `onRestore`, but it will be called after all reactives, observers, and render functions run, and after results are sent to the client browser. `onRestored` callbacks can be useful for sending update messages to the client browser.

Usage

```r
onBookmark(fun, session = getDefaultReactiveDomain())
onBookmarked(fun, session = getDefaultReactiveDomain())
onRestore(fun, session = getDefaultReactiveDomain())
onRestored(fun, session = getDefaultReactiveDomain())
```

Arguments

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

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

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

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

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

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

Modules

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

See Also

enableBookmarking for general information on bookmarking.

Examples

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

# Basic use of onBookmark and onRestore: This app saves the time in its
# arbitrary values, and restores that time when the app is restored.
ui <- function(req) {
  fluidPage(
    textInput("txt", "Input text"),
    bookmarkButton()
  )
}
server <- function(input, output) {
  onBookmark(function(state) {
    savedTime <- as.character(Sys.time())
    cat("Last saved at", savedTime, "\n")
    # state is a mutable reference object, and we can add arbitrary values to
    # it.
    state$values$time <- savedTime
  })
```
onBookmark

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

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

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

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

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

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

# This app has a module, and both the module and the main app code have
# onBookmark and onRestore functions which write and read state$values$hash. The
# module's version of state$values$hash does not conflict with the app's version
# of state$values$hash.
# A basic module that capitalizes text.
capitalizerUI <- function(id) {
  ns <- NS(id)
  wellPanel(
    h4("Text captializer module"),
    textInput(ns("text"), "Enter text:"),
    verbatimTextOutput(ns("out"))
  )
}
capitalizerServer <- function(input, output, session) {
  output$output <- renderText(
    toupper(input$text)
  )
  onBookmark(function(state) {
    state$values$hash <- rlang::hash(input$text)
  })
  onRestore(function(state) {
    if (identical(rlang::hash(input$text), state$values$hash)) {
      message("Module's input text matches hash ", state$values$hash)
    } else {
      message("Module's input text does not match hash ", state$values$hash)
    }
  })
}

# Main app code
ui <- function(request) {
  fluidPage(
    sidebarLayout(
      sidebarPanel(
        capitalizerUI("tc"),
        textInput("text", "Enter text (not in module):"),
        bookmarkButton()
      ),
      mainPanel()
    )
  )
}
server <- function(input, output, session) {
  callModule(capitalizerServer, "tc")
  onBookmark(function(state) {
    state$values$hash <- rlang::hash(input$text)
  })
  onRestore(function(state) {
    if (identical(rlang::hash(input$text), state$values$hash)) {
      message("App's input text matches hash ", state$values$hash)
    } else {
      message("App's input text does not match hash ", state$values$hash)
    }
  })
}

enableBookmarking(store = "url")
shinyApp(ui, server)
onFlush

Add callbacks for Shiny session events

Description

These functions are for registering callbacks on Shiny session events. onFlush registers a function that will be called before Shiny flushes the reactive system. onFlushed registers a function that will be called after Shiny flushes the reactive system. onUnhandledError registers a function to be called when an unhandled error occurs before the session is closed. onSessionEnded registers a function to be called after the client has disconnected.

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

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

Usage

onFlush(fun, once = TRUE, session = getDefaultReactiveDomain())
onFlushed(fun, once = TRUE, session = getDefaultReactiveDomain())
onSessionEnded(fun, session = getDefaultReactiveDomain())
onUnhandledError(fun, session = getDefaultReactiveDomain())

Arguments

fun
A callback function.

once
Should the function be run once, and then cleared, or should it re-run each time the event occurs. (Only for onFlush and onFlushed.)

session
A shiny session object.

Unhandled Errors

Unhandled errors are errors that aren’t otherwise handled by Shiny or by the application logic. In other words, they are errors that will either cause the application to crash or will result in "Error" output in the UI.

You can use onUnhandledError() to register a function that will be called when an unhandled error occurs. This function will be called with the error object as its first argument. If the error is fatal and will result in the session closing, the error condition will have the shiny.error.fatal class.

Note that the onUnhandledError() callbacks cannot be used to prevent the app from closing or to modify the error condition. Instead, they are intended to give you an opportunity to log the error or perform other cleanup operations.
See Also

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

Examples

```r
library(shiny)

ui <- fixedPage(
  markdown(c(
    "Set the number to 8 or higher to cause an error",
    "in the `renderText()` output."
  ),
  sliderInput("number", "Number", 0, 10, 4),
  textOutput("text"),
  hr(),
  markdown(c(
    "Click the button below to crash the app with an unhandled error",
    "in an `observe()` block."
  ),
  actionButton("crash", "Crash the app!"))
)

log_event <- function(level, ...) {
  ts <- strftime(Sys.time(), "[\%F \%T]\")
  message(level, ts, ...)
}

server <- function(input, output, session) {
  log_event("INFO", "Session started")

  onUnhandledError(function(err) {
    # log the unhandled error
    level <- if (inherits(err, "shiny.error.fatal")) "FATAL" else "ERROR"
    log_event(level, conditionMessage(err))
  })

  onStop(function() {
    log_event("INFO", "Session ended")
  })

  observeEvent(input$crash, stop("Oops, an unhandled error happened!"))

  output$text <- renderText({
    if (input$number > 7) {
      stop("that's too high!")
    }
    sprintf("You picked number %d.", input$number)
  })
}

shinyApp(ui, server)
```
onStop

Run code after an application or session ends

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

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

Arguments
- **fun**: A function that will be called after the app has finished running.
- **session**: A scope for when the callback will run. If `onStop` is called from within the server function, this will default to the current session, and the callback will be invoked when the current session ends. If `onStop` is called outside a server function, then the callback will be invoked with the application exits. If `NULL`, it is the same as calling `onStop` outside of the server function, and the callback will be invoked when the application exits.

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

See Also
- `onSessionEnded()` for the same functionality, but at the session level only.

Examples
```r
## Only run this example in interactive R sessions
if (interactive()) {
  # Open this application in multiple browsers, then close the browsers.
  shinyApp(
    ui = basicPage("onStop demo"),
    server = function(input, output, session) {
      onStop(function() cat("Session stopped\n"))
    },
    onStart = function() {
      cat("Doing application setup\n")
      onStop(function() {
        cat("Doing application cleanup\n")
      }()
    }
  )
}
```
# In the example above, onStop() is called inside of onStart(). This is
# the pattern that should be used when creating a shinyApp() object from
# a function, or at the console. If instead you are writing an app.R which
# will be invoked with runApp(), you can do it that way, or put the onStop()
# before the shinyApp() call, as shown below.

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

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

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

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

## End(Not run)

---

### outputOptions

Set options for an output object.

---

### Description

These are the available options for an output object:

- `suspendWhenHidden`. When `TRUE` (the default), the output object will be suspended (not
eexecute) 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.
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 as a nested list when it contains pairs of square brackets []. For example, the query ‘a[i1][j1]=x&b[i1][j1]=y&b[i2][j1]=z’ will be parsed as list(a = list(i1 = list(j1 = 'x')), b = list(i1 = list(j1 = 'y'), i2 = list(j1 = 'z'))) when nested = TRUE, and list(‘a[i1][j1]’ = ‘x’, ’b[i1][j1]’ = ‘y’, ’b[i2][j1]’ = ‘z’) when nested = FALSE.

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)
Examples

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

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

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

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

    # Return a string with key-value pairs
    paste(names(query), query, sep = "=" , collapse="", )
  })
}
## End(Not run)
```

### passwordInput

Create a password input control

**Description**

Create an password control for entry of passwords.

**Usage**

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

**Arguments**

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

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

Server value

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

See Also

updateTextInput()

Other input elements: `actionButton()`, `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `numericInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, `submitButton()`, `textareaInput()`, `textInput()`, `varSelectInput()

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    passwordInput("password", "Password:"),
    actionButton("go", "Go"),
    verbatimTextOutput("value")
  )
  server <- function(input, output) {
    output$value <- renderText({
      req(input$go)
      isolate(input$password)
    })
  }
  shinyApp(ui, server)
}
```

---

plotOutput

Create an plot or image output element

Description

Render a `renderPlot()` or `renderImage()` within an application page.

Usage

```r
imageOutput(
  outputId,
  width = "100%",
  height = "400px",
  click = NULL,
  dblclick = NULL,
```
```r
plotOutput(
  outputId,
  width = "100%",
  height = "400px",
  click = NULL,
  dblclick = NULL,
  hover = NULL,
  brush = NULL,
  inline = FALSE,
  fill = !inline
)
```

**Arguments**

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

`inline` use an inline (`span()`) or block container (`div()`) for the output

`fill` Whether or not the returned tag should be treated as a fill item, meaning that its height is allowed to grow/shrink to fit a fill container with an opinionated height (see `htmltools::bindFillRole()`) with an opinionated height. Examples of fill containers include `bslib::card()` and `bslib::card_body_fill()`.

**Value**

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

**Interactive plots**

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

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

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

**Note**

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

**See Also**

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

**Examples**

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

# A demonstration of clicking, hovering, and brushing
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4, height=300,
        plotOutput("plot", click = "plot_click", hover = hoverOpts(id = "plot_hover", delayType = "throttle"), brush = brushOpts(id = "plot_brush"),
          h4("Clicked points"),
          tableOutput("plot_clickedpoints"),
          h4("Brushed points"),
          tableOutput("plot_brushedpoints")
      ),
      column(width = 4,
        verbatimTextOutput("plot_clickinfo"),
        verbatimTextOutput("plot_hoverinfo")
      ),
      column(width = 4,
        wellPanel(actionButton("newplot", "New plot"),
                  verbatimTextOutput("plot_brushinfo")
      )
    ),
    server = function(input, output, session) {
      data <- reactive({
        input$newplot
        # Add a little noise to the cars data so the points move
        cars + rnorm(nrow(cars))
      })
      output$plot <- renderPlot({
        d <- data()
        plot(d$speed, d$dist)
      })
      output$plot_clickinfo <- renderPrint({
        cat("Click:
        str(input$plot_click)
```r
# Demo of clicking, hovering, brushing with imageOutput
# Note that coordinates are in pixels
shinyApp(
  ui = basicPage(
    fluidRow(
      column(width = 4, imageOutput("image", height=300, 
        click = "image_click",
        hover = hoverOpts( 
          id = "image_hover",
          delay = 500,
          delayType = "throttle"
        ),
        brush = brushOpts(id = "image_brush")
      ),
      column(width = 4, verbatimTextOutput("image_clickinfo"),
        verbatimTextOutput("image_hoverinfo")
      ),
      column(width = 4, wellPanel(actionButton("newimage", "New image")),
        verbatimTextOutput("image_brushinfo")
    )
  ),
)
)
```
server = function(input, output, session) {
  output$image <- renderImage({
    input$newimage

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

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

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

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

Description

Capture a plot as a PNG file.

The PNG graphics device used is determined in the following order:

- If the ragg package is installed (and the shiny.useragg is not set to FALSE), then use `ragg::agg_png()`.
- If a quartz device is available (i.e., `capabilities("aqua")` is TRUE), then use `png(type = "quartz")`. 
• If the Cairo package is installed (and the shiny.usecairo option is not set to FALSE), then use Cairo::CairoPNG().
• Otherwise, use grDevices::png(). In this case, Linux and Windows may not antialias some point shapes, resulting in poor quality output.

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 the graphics device. 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 the graphics device. These can be used to set the width, height, background color, etc.

Details

A NULL value provided to width or height is ignored (i.e., the default width or height of the graphics device is used).

Value

A path to the newly generated PNG file.

---

Progress Reporting progress (object-oriented API)

Description

Reporting progress (object-oriented API)
Reporting progress (object-oriented API)
Details

Reports progress to the user during long-running operations.

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

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

As of version 0.14, the progress indicators use Shiny’s new notification API. If you want to use the old styling (for example, you may have used customized CSS), you can use `style="old"` each time you call `Progress$new()`. If you don’t want to set the style each time `Progress$new` is called, you can instead call `shinyOptions(progress.style="old")` just once, inside the server function.

Methods

Public methods:

- `Progress$new()`
- `Progress$set()`
- `Progress$inc()`
- `Progress$getMin()`
- `Progress$getMax()`
- `Progress$getValue()`
- `Progress$close()
- `Progress$clone()`

Method `new()`: Creates a new progress panel (but does not display it).

Usage:

```r
Progress$new(
  session = getDefaultReactiveDomain(),
  min = 0,
  max = 1,
  style = getShinyOption("progress.style", default = "notification")
)
```

Arguments:

- `session` The Shiny session object, as provided by `shinyServer` to the server function.
- `min` The value that represents the starting point of the progress bar. Must be less than `max`.
- `max` The value that represents the end of the progress bar. Must be greater than `min`.
- `style` Progress display style. If "notification" (the default), the progress indicator will show using Shiny’s notification API. If "old", use the same HTML and CSS used in Shiny 0.13.2 and below (this is for backward-compatibility).

Method `set()`: Updates the progress panel. When called the first time, the progress panel is displayed.

Usage:

```r
Progress$set(value = NULL, message = NULL, detail = NULL)
```
Arguments:
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.

Method inc(): 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.
Usage:
Progress$inc(amount = 0.1, message = NULL, detail = NULL)
Arguments:
amount  For the inc() method, a numeric value to increment the progress bar.
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.

Method getMin(): Returns the minimum value.
Usage:
Progress$getMin()

Method getMax(): Returns the maximum value.
Usage:
Progress$getMax()

Method getValue(): Returns the current value.
Usage:
Progress$getValue()

Method close(): Removes the progress panel. Future calls to set and close will be ignored.
Usage:
Progress$close()

Method clone(): The objects of this class are cloneable with this method.
Usage:
Progress$clone(deep = FALSE)
Arguments:
deep  Whether to make a deep clone.

See Also
withProgress()
radioButtons

Examples

## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    plotOutput("plot")
  )

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

      progress$set(message = 'Calculation in progress',
        detail = 'This may take a while...')
      for (i in 1:15) {
        progress$set(value = i)
        Sys.sleep(0.5)
      }
      plot(cars)
    })
  }

  shinyApp(ui, server)
}

radioButtons
Create radio buttons

Description

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

Usage

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

inputId: The input slot that will be used to access the value.

label: Display label for the control, or NULL for no label.

choices: List of values to select from (if elements of the list are named then that name rather than the value is displayed to the user). If this argument is provided, then choiceNames and choiceValues must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings.

selected: The initially selected value. If not specified, then it defaults to the first item in choices. To start with no items selected, use character(0).

inline: If TRUE, render the choices inline (i.e. horizontally)

width: The width of the input, e.g. '400px', or '100%'; see validateCssUnit().

choiceNames, choiceValues: List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, choiceNames and choiceValues must have the same length). If either of these arguments is provided, then the other must be provided and choices must not be provided. The advantage of using both of these over a named list for choices is that choiceNames allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples.

Details

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

Value

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

Server value

A character string containing the value of the selected button.

See Also

updateRadioButtons()

Other input elements: actionButton(), checkboxGroupInput(), checkboxInput(), dateInput(), dateRangeInput(), fileInput(), numericInput(), passwordInput(), selectInput(), sliderInput(), submitButton(), textareaInput(), textInput(), varSelectInput()
Examples

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

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

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

      hist(dist(500))
    })
  }

  shinyApp(ui, server)

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

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

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

Create a reactive expression

**Description**

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

**Usage**

```r
reactive(
  x,
  env = parent.frame(),
  quoted = FALSE,
  ...,  # unused
  label = NULL,
  domain = getDefaultReactiveDomain(),
  ..stacktraceon = TRUE
)
```

```r
is.reactive(x)
```

**Arguments**

- **x**
  - For `is.reactive()`, an object to test. For `reactive()`, an expression. When passing in a `rlang::quo()` sure with `reactive()`, remember to use `rlang::inject()` to distinguish that you are passing in the content of your quosure, not the expression of the quosure.
- **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. If `x` is a quosure and quoted is TRUE, then env is ignored.
- **quoted**
  - If it is TRUE, then the `quote()`ed value of `x` will be used when `x` is evaluated. If `x` is a quosure and you would like to use its expression as a value for `x`, then you must set quoted to TRUE.
- **...**
  - Not used.
- **label**
  - A label for the reactive expression, useful for debugging.
- **domain**
  - See `domains`.
- **..stacktraceon**
  - Advanced use only. For stack manipulation purposes; see `stacktrace()`.

**Details**

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

See the [Shiny tutorial](https://shiny.rstudio.com) for more information about reactive expressions.
Value

a function, wrapped in a S3 class "reactive"

Examples

library(rlang)
values <- reactiveValues(A=1)

reactiveB <- reactive(
  values$A + 1
)
# View the values from the R console with isolate()
isolate(reactiveB())
# 2

# To store expressions for later conversion to reactive, use quote()
myquo <- rlang::quo(values$A + 2)
# Unexpected value! Sending a quosure directly will not work as expected.
reactiveC <- reactive(myquo)
# We'd hope for `3`, but instead we get the quosure that was supplied.
isolate(reactiveC())

# Instead, the quosure should be `rlang::inject()`ed
reactiveD <- rlang::inject(reactive(!!myquo))
isolate(reactiveD())
# 3

# (Legacy) Can use quoted expressions
expr <- quote({ values$A + 3 })
reactiveE <- reactive(expr, quoted = TRUE)
isolate(reactiveE())
# 4

---

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

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

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

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

## End(Not run)
```
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.
reactiveTimer

Timer

Description

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

Usage

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

Arguments

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

See Also

reactiveFileReader()
Details

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

See invalidateLater() as a safer and simpler alternative.

Value

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

See Also

invalidateLater()

Examples

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

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

server <- function(input, output) {

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

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

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

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

shinyApp(ui, server)
```
reactiveVal

Create a (single) reactive value

Description

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

Usage

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

Arguments

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

Details

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

Value

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

Examples

```r
## Not run:

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

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

# Read the value by calling without arguments
r()
```
## End(Not run)

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

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

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

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

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

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

shinyApp(ui, server)

--

**reactiveValues**

*Create an object for storing reactive values*

**Description**

This function returns an object for storing reactive values. It is similar to a list, but with special capabilities for reactive programming. When you read a value from it, the calling reactive expression takes a reactive dependency on that value, and when you write to it, it notifies any reactive functions that depend on that value. Note that values taken from the reactiveValues object are reactive, but the reactiveValues object itself is not.

**Usage**

reactiveValues(...)
reactiveValuesToList

Convert a reactivevalues object to a list

Description

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

Usage

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

- **x**  
  A reactivevalues object.

- **all.names**  
  If TRUE, include objects with a leading dot. If FALSE (the default) don’t include those objects.

Examples

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

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

---

**reactlog**  
*Reactive Log Visualizer*

Description

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

Usage

- `reactlog()`
- `reactlogShow(time = TRUE)`
- `reactlogReset()`
- `reactlogAddMark(session = getDefaultReactiveDomain())`

Arguments

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

- **session**  
  The Shiny session to assign the mark to. Defaults to the current session.
Details

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

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

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

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

For security and performance reasons, do not enable options(`shiny.reactlog=TRUE`) (or `reactlog::reactlog_enable()`) 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. In addition, reactlog should be considered a memory leak as it will constantly grow and will never reset until the R session is restarted.

Functions

- `reactlog()`: Return a list of reactive information. Can be used in conjunction with `reactlog::reactlog_show` to later display the reactlog graph.
- `reactlogShow()`: Display a full reactlog graph for all sessions.
- `reactlogReset()`: Resets the entire reactlog stack. Useful for debugging and removing all prior reactive history.
- `reactlogAddMark()`: Adds "mark" entry into the reactlog stack. This is useful for programmatically adding a marked entry in the reactlog, rather than using your keyboard's key combination.

For example, we can mark the reactlog at the beginning of an `observeEvent`'s calculation:

```r
observeEvent(input$my_event_trigger, {
    # Add a mark in the reactlog
    reactlogAddMark()
    # Run your regular event reaction code here...
    ....
})
```
registerInputHandler

Register an Input Handler

Description

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

Usage

 registerInputHandler(type, fun, force = FALSE)

Arguments

 type The type for which the handler should be added — should be a single-element character vector.
 fun The handler function. This is the function that will be used to parse the data delivered from the client before it is available in the input variable. The function will be called with the following three parameters:
   1. The value of this input as provided by the client, deserialized using jsonlite.
   2. The shinysession in which the input exists.
   3. The name of the input.
 force If TRUE, will overwrite any existing handler without warning. If FALSE, will throw an error if this class already has a handler defined.

Details

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

 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() applyInputHandlers()
removeInputHandler

Deregister an Input Handler

Description

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

Usage

removeInputHandler(type)

Arguments

type

The type for which handlers should be removed.

Value

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

See Also

registerInputHandler()
Description

Renders a reactive plot, with plot images cached to disk. As of Shiny 1.6.0, this is a shortcut for using `bindCache()` with `renderPlot()`.

Usage

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

Arguments

- **expr**: An expression that generates a plot.
- **cacheKeyExpr**: An expression that returns a cache key. This key should be a unique identifier for a plot: the assumption is that if the cache key is the same, then the plot will be the same.
- **sizePolicy**: A function that takes two arguments, `width` and `height`, and returns a list with `width` and `height`. The purpose is to round the actual pixel dimensions from the browser to some other dimensions, so that this will not generate and cache images of every possible pixel dimension. See `sizeGrowthRatio()` for more information on the default sizing policy.
- **res**: The resolution of the PNG, in pixels per inch.
- **cache**: The scope of the cache, or a cache object. This can be "app" (the default), "session", or a cache object like a `cachem::cache_disk()`. See the Cache Scoping section for more information.
- **...**: Arguments to be passed through to `plotPNG()`. These can be used to set the width, height, background color, etc.
- **alt**: Alternate text for the HTML `<img>` tag if it cannot be displayed or viewed (i.e., the user uses a screen reader). In addition to a character string, the value may be a reactive expression (or a function referencing reactive values) that returns a character string. If the value is `NA` (the default), then `ggplot2::get_alt_text()` is used to extract alt text from ggplot objects; for other plots, `NA` results in alt
renderCachedPlot

text of "Plot object". NULL or "" is not recommended because those should be limited to decorative images.

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

width, height not used. They are specified via the argument sizePolicy.

Details

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

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

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

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

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

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

Interactive plots

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

See Also

See renderPlot() for the regular, non-cached version of this function. It can be used with bindCache() to get the same effect as renderCachedPlot(). For more about configuring caches, see cachem::cache_mem() and cachem::cache_disk().

Examples

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

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

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

ui <- fluidPage(
  sidebarLayout(
    sliderInput("n", "Number of points", 50, 400, 100, step = 50),
    actionButton("newdata", "New data")
  ),
  mainPanel(
    plotOutput("plot")
  )
)

server <- function(input, output, session) {
  observeEvent(input$newdata, {
    mydata(data.frame(x = rnorm(400), y = rnorm(400)))
  })
  output$plot <- renderCachedPlot(
    Sys.sleep(2)
    d <- mydata()
seqn <- seq_len(input$n)
plot(d$x[seqn], d$y[seqn], xlim = range(d$x), ylim = range(d$y))
},
  cacheKeyExpr = { list(input$n, mydata()) },
)
}

shinyApp(ui, server)

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

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

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

Image file output

**Description**

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

**Usage**

```r
renderImage(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  deleteFile,
  outputArgs = list()
)
```

**Arguments**

- `expr` An expression that returns a list.
- `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. If `expr` is a quosure and `quoted` is TRUE, then `env` is ignored.
renderImage

quoted

If it is TRUE, then the quote()ed value of expr will be used when expr is evaluated. If expr is a quosure and you would like to use its expression as a value for expr, then you must set quoted to TRUE.

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. (For backward compatibility reasons, if this argument is missing, a warning will be emitted, and if the file is in the temp directory it will be deleted. In the future, this warning will become an error.)

outputArgs

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

Details

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

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

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

See Also

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

Examples

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

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

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

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

      # Generate a png
      png(outfile, width=400, height=400)

      # Your code here, which generates a plot
    })
  }
}
renderPlot

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

Usage

renderPlot(
    expr,
    width = "auto",
    height = "auto",
    res = 72,
    ...,
    alt = NA,
    env = parent.frame(),
    quoted = FALSE,
    execOnResize = FALSE,
    outputArgs = list()
)

Arguments

expr  An expression that generates a plot.

width, height Height and width can be specified in three ways:

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

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

res Resolution of resulting plot, in pixels per inch. This value is passed to `plotPNG()`. 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 `plotPNG()`. These can be used to set the width, height, background color, etc.

alt Alternate text for the HTML `<img>` tag if it cannot be displayed or viewed (i.e., the user uses a screen reader). In addition to a character string, the value may be a reactive expression (or a function referencing reactive values) that returns a character string. If the value is NA (the default), then `ggplot2::get_alt_text()` is used to extract alt text from ggplot objects; for other plots, NA results in alt text of "Plot object". NULL or "" is not recommended because those should be limited to decorative images.

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. If `expr` is a quosure and quoted is TRUE, then env is ignored.

quoted If it is TRUE, then the `quote()`ed value of expr will be used when expr is evaluated. If `expr` is a quosure and you would like to use its expression as a value for expr, then you must set quoted to TRUE.
execOnResize If FALSE (the default), then when a plot is resized, Shiny will replay the plot drawing commands with `grDevices::replayPlot()` instead of re-executing `expr`. This can result in faster plot redrawing, but there may be rare cases where it is undesirable. If you encounter problems when resizing a plot, you can have Shiny re-execute the code on resize by setting this to TRUE.

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

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

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

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

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

renderPrint

Text Output

Description
renderPrint() prints the result of `expr`, while `renderText()` pastes it together into a single string. `renderPrint()` is equivalent to `print()`; `renderText()` is equivalent to `cat()`. Both functions capture all other printed output generated while evaluating `expr`.

renderPrint() is usually paired with `verbatimTextOutput()`; `renderText()` is usually paired with `textOutput()`.

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

renderText(
renderPrint

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

Arguments

expr                     An expression to evaluate.
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. If
                         expr is a quosure and quoted is TRUE, then env is ignored.
quoted                   If it is TRUE, then the quote()ed value of expr will be used when expr is evalu-
                         ated. If expr is a quosure and you would like to use its expression as a value
                         for expr, then you must set quoted to TRUE.
width                    Width of printed output.
outputArgs               A list of arguments to be passed through to the implicit call to verbatimTextOutput() or
textOutput() when the functions are used in an interactive RMarkdown document.
sep                      A separator passed to cat to be appended after each element.

Details

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

Value

For renderPrint(), note the given expression returns NULL then NULL will actually be visible in
the output. To display nothing, make your function return invisible().

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"
    )
)

renderUI

})

multiprintFun()
# ']

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

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

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

})
Description

Renders reactive HTML using the Shiny UI library.

Usage

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

Arguments

expr          An expression that returns a Shiny tag object, HTML(), or a list of such objects.
env           The parent environment for the reactive expression. By default, this is the calling
              environment, the same as when defining an ordinary non-reactive expression. If
              expr is a quosure and quoted is TRUE, then env is ignored.
quoted        If it is TRUE, then the quote()ed value of expr will be used when expr is evalu-
              ated. If expr is a quosure and you would like to use its expression as a value
              for expr, then you must set quoted to TRUE.
outputArgs    A list of arguments to be passed through to the implicit call to uiOutput() when
              renderUI is used in an interactive R Markdown document.

Details

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

See Also

uiOutput()

Examples

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

  ui <- fluidPage(
    uiOutput("moreControls")
  )

  server <- function(input, output) {
    output$moreControls <- renderUI({
      tagList(
        sliderInput("n", "N", 1, 1000, 500),
        textInput("label", "Label")
      )
    })
  }

  shinyApp(ui, server)
```

**repeatable**  

*Make a random number generator repeatable*

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

A repeatable version of the function that was passed in.

**Note**

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

**Examples**

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

**req**  

*Check for required values*

**Description**

Ensure that values are available ("truthy") before proceeding with a calculation or action. If any of the given values is not truthy, the operation is stopped by raising a "silent" exception (not logged by Shiny, nor displayed in the Shiny app's UI).
Usage

req(..., cancelOutput = FALSE)

Arguments

...  
Values to check for truthiness.
cancelOutput  
If TRUE and an output is being evaluated, stop processing as usual but instead of clearing the output, leave it in whatever state it happens to be in. If "progress", do the same as TRUE, but also keep the output in recalculating state; this is intended for cases when an in-progress calculation will not be completed in this reactive flush cycle, but is still expected to provide a result in the future.

Details

The req function was designed to be used in one of two ways. The first is to call it like a statement (ignoring its return value) before attempting operations using the required values:

```r
rv <- reactiveValues(state = FALSE)
r <- reactive(
  req(input$a, input$b, rv$state)
  # Code that uses input$a, input$b, and/or rv$state...
)
```

In this example, if r() is called and any of input$a, input$b, and rv$state are NULL, FALSE, "", etc., then the req call will trigger an error that propagates all the way up to whatever render block or observer is executing.

The second is to use it to wrap an expression that must be truthy:

```r
output$plot <- renderPlot({
  if (req(input$plotType) == "histogram") {
    hist(dataset())
  } else if (input$plotType == "scatter") {
    qplot(dataset(), aes(x = x, y = y))
  }
})
```

In this example, req(input$plotType) first checks that input$plotType is truthy, and if so, returns it. This is a convenient way to check for a value "inline" with its first use.

Value

The first value that was passed in.

Using req(FALSE)

You can use req(FALSE) (i.e. no condition) if you’ve already performed all the checks you needed to by that point and just want to stop the reactive chain now. There is no advantage to this, except perhaps ease of readability if you have a complicated condition to check for (or perhaps if you’d like to divide your condition into nested if statements).
Using `cancelOutput = TRUE`

When `req(..., cancelOutput = TRUE)` is used, the "silent" exception is also raised, but it is treated slightly differently if one or more outputs are currently being evaluated. In those cases, the reactive chain does not proceed or update, but the output(s) are left in whatever state they happen to be in (whatever was their last valid state).

Note that this is always going to be the case if this is used inside an output context (e.g. `output$txt <- ...`). It may or may not be the case if it is used inside a non-output context (e.g. `reactive()`, `observe()` or `observeEvent()`) — depending on whether or not there is an `output$...` that is triggered as a result of those calls. See the examples below for concrete scenarios.

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    textInput('data', 'Enter a dataset from the "datasets" package', 'cars'),
    p('E.g. "cars", "mtcars", "pressure", "faithful"'), hr(),
    tableOutput('tbl')
  )

  server <- function(input, output) {
    output$tbl <- renderTable({
      ## to require that the user types something, use: `req(input$data)`
      ## but better: require that input$data is valid and leave the last
      ## valid table up
      req(exists(input$data, "package:datasets", inherits = FALSE),
          cancelOutput = TRUE)

      head(get(input$data, "package:datasets", inherits = FALSE))
    })
  }

  shinyApp(ui, server)
}
```

---

`restoreInput`  
*Restore an input value*

**Description**

This restores an input value from the current restore context. It should be called early on inside of input functions (like `textInput()`).

**Usage**

`restoreInput(id, default)`
Arguments

**id**  
Name of the input value to restore.

**default**  
A default value to use, if there's no value to restore.

---

**runApp**  
**Run Shiny Application**

Description

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

Usage

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

Arguments

**appDir**  
The application to run. Should be one of the following:

- A directory containing `server.R`, plus, either `ui.R` or a `www` directory that contains the file `index.html`.
- A directory containing `app.R`.
- An `.R` file containing a Shiny application, ending with an expression that produces a Shiny app object.
- A list with `ui` and `server` components.
- A Shiny app object created by `shinyApp()`.

**port**  
The TCP port that the application should listen on. If the `port` is not specified, and the `shiny.port` option is set (with `options(shiny.port = XX)`), then that port will be used. Otherwise, use a random port between 3000:8000, excluding ports that are blocked by Google Chrome for being considered unsafe: 3659, 4045, 5060, 5061, 6000, 6566, 6665:6669 and 6697. Up to twenty random ports will be tried.

**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. The value of this parameter can also be a function to call with the application's URL.
**runApp**  

The IPv4 address that the application should listen on. Defaults to the `shiny.host` option, if set, or "127.0.0.1" if not. See Details.

**workerId**  
Can generally be ignored. Exists to help some editions of Shiny Server Pro route requests to the correct process.

**quiet**  
Should Shiny status messages be shown? Defaults to FALSE.

**display.mode**  
The mode in which to display the application. If set to the value "showcase", shows application code and metadata from a DESCRIPTION file in the application directory alongside the application. If set to "normal", displays the application normally. Defaults to "auto", which displays the application in the mode given in its DESCRIPTION file, if any.

**test.mode**  
Should the application be launched in test mode? This is only used for recording or running automated tests. Defaults to the `shiny.testmode` option, or FALSE if the option is not set.

**Details**

The host parameter was introduced in Shiny 0.9.0. Its default value of "127.0.0.1" means that, contrary to previous versions of Shiny, only the current machine can access locally hosted Shiny apps. To allow other clients to connect, use the value "0.0.0.0" instead (which was the value that was hard-coded into Shiny in 0.8.0 and earlier).

**Examples**

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

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

## End(Not run)

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

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

  # Running a Shiny app object
  app <- shinyApp(
```
runExample

Description

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

Usage

runExample(
  example = NA,
  port = getOption("shiny.port"),
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"),
  display.mode = c("auto", "normal", "showcase"),
  package = "shiny"
)

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. If the port is not specified, and the shiny.port option is set (with options(shiny.port = XX)), then that port will be used. Otherwise, use a random port between 3000:8000, excluding ports that are blocked by Google Chrome for being considered unsafe: 3659, 4045, 5060, 5061, 6000, 6566, 6665:6669 and 6697. Up to twenty random ports will be tried.

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.
package

The package in which to find the example (defaults to "shiny"). To provide examples in your package, store examples in the `inst/examples-shiny` directory of your package. Each example should be in its own subdirectory and should be runnable when `runApp()` is called on the subdirectory. Example apps can include a DESCRIPTION file and a README.md file to provide metadata and commentary about the example. See the article on Display Modes on the Shiny website for more information.

Examples

```r
## Only run this example in interactive R sessions
if (interactive()) {
  # List all available examples
  runExample()

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

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

---

**runGadget**

*Run a gadget*

**Description**

Similar to runApp, but handles input$cancel automatically, and if running in RStudio, defaults to viewing the app in the Viewer pane.

**Usage**

```r
runGadget(
  app,
  server = NULL,
  port =getOption("shiny.port"),
  viewer = paneViewer(),
  stopOnCancel = TRUE
)
```

**Arguments**

- **app**
  - Either a Shiny app object as created by `shinyApp()` et al, or, a UI object.
- **server**
  - Ignored if app is a Shiny app object; otherwise, passed along to `shinyApp` (i.e. `shinyApp(ui = app, server = server)`).
- **port**
  - See `runApp()`.
- **viewer**
  - Specify where the gadget should be displayed—viewer pane, dialog window, or external browser—by passing in a call to one of the `viewer()` functions.
stopOnCancel If TRUE (the default), then an observeEvent is automatically created that handles input$cancel by calling stopApp() with an error. Pass FALSE if you want to handle input$cancel yourself.

Value

The value returned by the gadget.

Examples

```r
## Not run:
library(shiny)
ui <- fillPage(...)

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

# Either pass ui/server as separate arguments...
runGadget(ui, server)

# ...or as a single app object
runGadget(shinyApp(ui, server))

## End(Not run)
```

---

runTests Runs the tests associated with this Shiny app

Description

Sources the .R files in the top-level of tests/ much like R CMD check. These files are typically simple runners for tests nested in other directories under tests/.

Usage

```r
runTests(appDir = ".", filter = NULL, assert = TRUE, envir = globalenv())
```

Arguments

- **appDir** The base directory for the application.
- **filter** If not NULL, only tests with file names matching this regular expression will be executed. Matching is performed on the file name including the extension.
- **assert** Logical value which determines if an error should be thrown if any error is captured.
- **envir** Parent testing environment in which to base the individual testing environments.
runUrl

Details

Historically, `shinytest` recommended placing tests at the top-level of the `tests/` directory. This older folder structure is not supported by `runTests`. Please see `shinyAppTemplate()` for more details.

Value

A data frame classed with the supplemental class "shiny_runtests". The data frame has the following columns:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>character(1)</td>
<td>File name of the runner script in <code>tests/</code> that was sourced.</td>
</tr>
<tr>
<td>pass</td>
<td>logical(1)</td>
<td>Whether or not the runner script signaled an error when sourced.</td>
</tr>
<tr>
<td>result</td>
<td>any or NA</td>
<td>The return value of the runner</td>
</tr>
</tbody>
</table>

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

```r
runUrl(url, filetype = NULL, subdir = NULL, destdir = NULL, ...)
runGist(gist, destdir = NULL, ...)
runGitHub(
  repo,
  username =getOption("github.user"),
  ref = "HEAD",
  subdir = NULL,
  destdir = NULL,
  ...
)
```
Arguments

url URL of the application.

filetype The file type (".zip", ".tar", or ".tar.gz"). Defaults to the file extension taken from the url.

subdir A subdirectory in the repository that contains the app. By default, this function will run an app from the top level of the repo, but you can use a path such as "inst/shinyapp".

destdir Directory to store the downloaded application files. If NULL (the default), the application files will be stored in a temporary directory and removed when the app exits.

... Other arguments to be passed to runApp(), such as port and launch.browser.

gist The identifier of the gist. For example, if the gist is https://gist.github.com/jcheng5/3239667, then 3239667, '3239667', and 'https://gist.github.com/jcheng5/3239667' are all valid values.

repo Name of the repository.

username GitHub username. If repo is of the form "username/repo", username will be taken from repo.

ref Desired git reference. Could be a commit, tag, or branch name. Defaults to "HEAD", which means the default branch on GitHub, typically "main" or "master".

Examples

```r
## Only run this example in interactive R sessions
if (interactive()) {
    runUrl("https://github.com/rstudio/shiny_example/archive/main.tar.gz")
}
```

```r
## Only run this example in interactive R sessions
if (interactive()) {
    runUrl("https://github.com/rstudio/shiny_example/archive/main.zip",
           subdir = "inst/shinyapp")
}
```

```r
## Only run this example in interactive R sessions
if (interactive()) {
    runGist(3239667)
    runGist("https://gist.github.com/jcheng5/3239667")
    runGist("https://gist.github.com/3239667")
}
```

```r
## Only run this example in interactive R sessions
if (interactive()) {
    runGitHub("shiny_example", "rstudio")
    # or runGitHub("rstudio/shiny_example")
    runGitHub("shiny_example", "rstudio", subdir = "inst/shinyapp")
}
```
safeError

Declare an error safe for the user to see

Description

This should be used when you want to let the user see an error message even if the default is to sanitize all errors. If you have an error e and call stop(safeError(e)), then Shiny will ignore the value of getOption("shiny.sanitize.errors") and always display the error in the app itself.

Usage

safeError(error)

Arguments

error

Either an "error" object or a "character" object (string). In the latter case, the string will become the message of the error returned by safeError.

Details

An error generated by safeError has priority over all other Shiny errors. This can be dangerous. For example, if you have set options(shiny.sanitize.errors = TRUE), then by default all error messages are omitted in the app, and replaced by a generic error message. However, this does not apply to safeError: whatever you pass through error will be displayed to the user. So, this should only be used when you are sure that your error message does not contain any sensitive information. In those situations, safeError can make your users’ lives much easier by giving them a hint as to where the error occurred.

Value

An "error" object

See Also

shiny-options()

Examples

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

# uncomment the desired line to experiment with shiny.sanitize.errors
# options(shiny.sanitize.errors = TRUE)
# options(shiny.sanitize.errors = FALSE)

# Define UI
ui <- fluidPage(
  textInput('number', 'Enter your favorite number from 1 to 10', '5'),
  textOutput('normalError'),
```
textOutput('safeError')
}

# Server logic
server <- function(input, output) {
  output$normalError <- renderText({
    number <- input$number
    if (number %in% 1:10) {
      return(paste('You chose', number, '!'))
    } else {
      stop(
        paste(number, 'is not a number between 1 and 10')
      )
    }
  })
  output$safeError <- renderText({
    number <- input$number
    if (number %in% 1:10) {
      return(paste('You chose', number, '!'))
    } else {
      stop(safeError(
        paste(number, 'is not a number between 1 and 10')
      ))
    }
  })
}

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

---

**selectInput**

Create a select list input control

**Description**

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

**Usage**

```r
selectInput(
  inputId,
  label,
  choices,
  selected = NULL,
  multiple = FALSE,
  selectize = TRUE,
  width = NULL,
  size = NULL
)```

selectInput(inputId, ..., options = NULL, width = NULL)

Arguments

inputId The input slot that will be used to access the value.
label Display label for the control, or NULL for no label.
choices List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the group labels (leveraging the <optgroup> HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature.
selected The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
multiple Is selection of multiple items allowed?
selectize Whether to use selectize.js or not.
width The width of the input, e.g. '400px', or '100%'; see validateCssUnit().
size Number of items to show in the selection box; a larger number will result in a taller box. Not compatible with selectize=TRUE. Normally, when multiple=FALSE, a select input will be a drop-down list, but when size is set, it will be a box instead.
... Arguments passed to selectInput().
options A list of options. See the documentation of selectize.js(https://selectize.dev/docs/usage) for possible options (character option values inside base::I() will be treated as literal JavaScript code; see renderDataTable() for details).

Details

By default, selectInput() and selectizeInput() use the JavaScript library selectize.js(https://selectize.dev/) instead of the basic select input element. To use the standard HTML select input element, use selectInput() with selectize=FALSE.

In selectize mode, if the first element in choices has a value of "", its name will be treated as a placeholder prompt. For example: selectInput("letter", "Letter", c("Choose one" = "", LETTERS))

Performance note: selectInput() and selectizeInput() can slow down significantly when thousands of choices are used; with legacy browsers like Internet Explorer, the user interface may hang for many seconds. For large numbers of choices, Shiny offers a "server-side selectize" option that massively improves performance and efficiency; see this selectize article on the Shiny Dev Center for details.

Value

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

A vector of character strings, usually of length 1, with the value of the selected items. When `multiple=TRUE` and nothing is selected, this value will be `NULL`.

Note

The selectize input created from `selectizeInput()` allows deletion of the selected option even in a single select input, which will return an empty string as its value. This is the default behavior of `selectize.js`. However, the selectize input created from `selectInput(..., selectize = TRUE)` will ignore the empty string value when it is a single choice input and the empty string is not in the `choices` argument. This is to keep compatibility with `selectInput(..., selectize = FALSE)`.

See Also

`updateSelectInput()` `varSelectInput()`

Other input elements: `actionButton()`, `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `numericInput()`, `passwordInput()`, `radioButtons()`, `sliderInput()`, `submitButton()`, `textareaInput()`, `textInput()`, `varSelectInput()`

Examples

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

# basic example
shinyApp(
  ui = fluidPage(
    selectInput("variable", "Variable: ",
      c("Cylinders" = "cyl",
      "Transmission" = "am",
      "Gears" = "gear")),
    tableOutput("data"),
  ),
  server = function(input, output) {
    output$data <- renderTable({
      mtcars[, c("mpg", input$variable), drop = FALSE]
    }, rownames = TRUE)
  }
)

# demoing group support in the `choices` arg
shinyApp(
  ui = fluidPage(
    selectInput("state", "Choose a state: ",
      list("East Coast" = list("NY", "NJ", "CT"),
      "West Coast" = list("WA", "OR", "CA"),
      "Midwest" = list("MN", "WI", "IA"))),
    textOutput("result"),
  ),
  server = function(input, output) {
    ```
serverInfo

Collect information about the Shiny Server environment

Description

This function returns the information about the current Shiny Server, such as its version, and whether it is the open source edition or professional edition. If the app is not served through the Shiny Server, this function just returns list(shinyServer = FALSE).

Usage

serverInfo()

Details

This function will only return meaningful data when using Shiny Server version 1.2.2 or later.

Value

A list of the Shiny Server information.

session

Session object

Description

Shiny server functions can optionally include session as a parameter (e.g. function(input, output, session)). The session object is an environment that can be used to access information and functionality relating to the session. The following list describes the items available in the environment; they can be accessed using the $ operator (for example, session$clientData$url_search).
Value

allowReconnect(value)

If value is TRUE and run in a hosting environment (Shiny Server or Connect) with reconnections enabled, then when the session ends due to the network connection closing, the client will attempt to reconnect to the server. If a reconnection is successful, the browser will send all the current input values to the new session on the server, and the server will recalculate any outputs and send them back to the client. If value is FALSE, reconnections will be disabled (this is the default state). If "force", then the client browser will always attempt to reconnect. The only reason to use "force" is for testing on a local connection (without Shiny Server or Connect).

clientData
A reactiveValues() object that contains information about the client.

- pixelratio reports the "device pixel ratio" from the web browser, or 1 if none is reported. The value is 2 for Apple Retina displays.
- singletons - for internal use
- url_protocol, url_hostname, url_port, url_pathname, url_search, url_hash_initial and url_hash can be used to get the components of the URL that was requested by the browser to load the Shiny app page. These values are from the browser's perspective, so neither HTTP proxies nor Shiny Server will affect these values. The url_search value may be used with parseQueryString() to access query string parameters.

clientData also contains information about each output. output_outputId_width and output_outputId_height give the dimensions (using offsetWidth and offsetHeight) of the DOM element that is bound to outputId, and output_outputId_hidden is a logical that indicates whether the element is hidden. These values may be NULL if the output is not bound.

input
The session's input object (the same as is passed into the Shiny server function as an argument).

isClosed()
A function that returns TRUE if the client has disconnected.

ns(id)
Server-side version of ns <- NS(id). If bare IDs need to be explicitly name-paced for the current module, session$ns("name") will return the fully-qualified ID.

onEnded(callback)
Synonym for onSessionEnded.

onFlush(func, once=TRUE)
Registers a function to be called before the next time (if once=TRUE) or every time (if once=FALSE) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

onFlushed(func, once=TRUE)
Registers a function to be called after the next time (if once=TRUE) or every time (if once=FALSE) Shiny flushes the reactive system. Returns a function that can be called with no arguments to cancel the registration.

onSessionEnded(callback)
Registers a function to be called after the client has disconnected. Returns a function that can be called with no arguments to cancel the registration.
output: The session’s output object (the same as is passed into the Shiny server function as an argument).

reactlog: For internal use.

registerDataObj(name, data, filterFunc)
Publishes any R object as a URL endpoint that is unique to this session. name must be a single element character vector; it will be used to form part of the URL. filterFunc must be a function that takes two arguments: data (the value that was passed into registerDataObj) and req (an environment that 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.

reload(): The equivalent of hitting the browser’s Reload button. Only works if the session is actually connected.

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

userData: An environment for app authors and module/package authors to store whatever session-specific data they want.

user: User’s log-in information. Useful for identifying users on hosted platforms such as RStudio Connect and Shiny Server.

groups: The user’s relevant group information. Useful for determining what privileges the user should or shouldn’t have.

resetBrush(brushId): Resets/clears the brush with the given brushId, if it exists on any imageOutput or plotOutput in the app.

sendCustomMessage(type, message)
Sends a custom message to the web page. type must be a single-element character vector giving the type of message, while message can be any jsonlite-encodable value. Custom messages have no meaning to Shiny itself; they are used solely 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, function(message){...}) as the page loads; the function you provide to addCustomMessageHandler will be invoked each time sendCustomMessage is called on the server.

sendBinaryMessage(type, message)
Similar to sendCustomMessage, but the message must be a raw vector and the registration method on the client is Shiny.addBinaryMessageHandler(type, function(message){...}). The message argument on the client will be a DataView.

sendInputMessage(inputId, message)
Sends a message to an input on the session’s client web page; if the input is present and bound on the page at the time the message is received, then the input binding object’s receiveMessage(el, message) method will be called. sendInputMessage should generally not be called directly from Shiny apps, but through friendlier wrapper functions like updateTextInput().

setBookmarkExclude(names)
Set input names to be excluded from bookmarking.
setBookmarkExclude

Exclude inputs from bookmarking

Description

This function tells Shiny which inputs should be excluded from bookmarking. It should be called from inside the application’s server function.

Usage

setBookmarkExclude(names = character(0), session = getDefaultReactiveDomain())
**setSerializer**

Arguments

- **names** A character vector containing names of inputs to exclude from bookmarking.
- **session** A shiny session object.

Details

This function can also be called from a module’s server function, in which case it will exclude inputs with the specified names, from that module. It will not affect inputs from other modules or from the top level of the Shiny application.

See Also

- enableBookmarking() for examples.

---

**setSerializer**

Add a function for serializing an input before bookmarking application state

---

**Description**

Add a function for serializing an input before bookmarking application state

**Usage**

```r
setSerializer(inputId, fun, session = getDefaultReactiveDomain())
```

**Arguments**

- **inputId** Name of the input value.
- **fun** A function that takes the input value and returns a modified value. The returned value will be used for the test snapshot.
- **session** A Shiny session object.

---

**shinyApp**

Create a Shiny app object

---

**Description**

These functions create Shiny app objects from either an explicit UI/server pair (shinyApp), or by passing the path of a directory that contains a Shiny app (shinyAppDir).
Usage

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

shinyAppDir(appDir, options = list())

shinyAppFile(appFile, options = list())

Arguments

ui The UI definition of the app (for example, a call to fluidPage() with nested controls).
If bookmarking is enabled (see enableBookmarking), this must be a single argument function that returns the UI definition.

server A function with three parameters: input, output, and session. The function is called once for each session ensuring that each app is independent.

onStart A function that will be called before the app is actually run. This is only needed for shinyAppObj, since in the shinyAppDir case, a global.R file can be used for this purpose.

options Named options that should be passed to the runApp call (these can be any of the following: "port", "launch.browser", "host", "quiet", "display.mode" and "test.mode"). You can also specify width and height parameters which 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.

enableBookmarking Can be one of "url", "server", or "disable". The default value, NULL, will respect the setting from any previous calls to enableBookmarking(). See enableBookmarking() for more information on bookmarking your app.

appDir Path to directory that contains a Shiny app (i.e. a server.R file and either ui.R or www/index.html)

appFile Path to a .R file containing a Shiny application

Details

Normally when this function is used at the R console, the Shiny app object is automatically passed to the print() function, which runs the app. If this is called in the middle of a function, the value
will not be passed to print() and the app will not be run. To make the app run, pass the app object to print() or runApp().

Value
An object that represents the app. Printing the object or passing it to runApp() will run the app.

Examples
### Only run this example in interactive R sessions
if (interactive()) {
  options(device.ask.default = FALSE)

  shinyApp(
    ui = fluidPage(
      numericInput("n", "n", 1),
      plotOutput("plot")
    ),
    server = function(input, output) {
      output$plot <- renderPlot( plot(head(cars, input$n)) )
    }
  )
}

shinyAppDir(system.file("examples/01_hello", package="shiny"))

# The object can be passed to runApp()
app <- shinyApp(
  ui = fluidPage(
    numericInput("n", "n", 1),
    plotOutput("plot")
  ),
  server = function(input, output) {
    output$plot <- renderPlot( plot(head(cars, input$n)) )
  }
)

runApp(app)

shinyAppTemplate  Generate a Shiny application from a template

Description
This function populates a directory with files for a Shiny application.

Usage
shinyAppTemplate(path = NULL, examples = "default", dryrun = FALSE)
shinyAppTemplate

Arguments

path
Path to create new shiny application template.

examples
Either one of "default", "ask", "all", or any combination of "app", "rdir", "module", and "tests". In an interactive session, "default" falls back to "ask"; in a non-interactive session, "default" falls back to "all". With "ask", this function will prompt the user to select which template items will be added to the new app directory. With "all", all template items will be added to the app directory.

dryrun
If TRUE, don’t actually write any files; just print out which files would be written.

Details

In an interactive R session, this function will, by default, prompt the user to select which components to add to the application. Choices are

1: All
2: app.R : Main application file
3: R/example.R : Helper file with R code
4: R/example-module.R : Example module
5: tests/testthat/ : Tests using the testthat and shinytest2 package

If option 1 is selected, the full example application including the following files and directories is created:

```
appdir/
| `-- app.R
|   `-- R
|     | `-- example-module.R
|     `-- example.R
`-- tests
    | `-- testthat.R
    `-- testthat
         | `-- setup-shinytest2.R
         | `-- test-examplemodule.R
         | `-- test-server.R
         | `-- test-shinytest2.R
         `-- test-sort.R
```

Some notes about these files:

- app.R is the main application file.
- All files in the R/ subdirectory are automatically sourced when the application is run.
- R/example.R and R/example-module.R are automatically sourced when the application is run. The first contains a function lexical_sort(), and the second contains code for module created by the moduleServer() function, which is used in the application.
- tests/ contains various tests for the application. You may choose to use or remove any of them. They can be executed by the runTests() function.
- `tests/testthat.R` is a test runner for test files in the `tests/testthat/` directory using the `shinytest2` package.
- `tests/testthat/setup-shinytest2.R` is setup file to source your `.R` folder into the testing environment.
- `tests/testthat/test-examplemodule.R` is a test for an application's module server function.
- `tests/testthat/test-server.R` is a test for the application's server code.
- `tests/testthat/test-shinytest2.R` is a test that uses the `shinytest2` package to do snapshot-based testing.
- `tests/testthat/test-sort.R` is a test for a supporting function in the R/ directory.

---

**showBookmarkUrlModal**

*Display a modal dialog for bookmarking*

**Description**

This is a wrapper function for `urlModal()` that is automatically called if an application is bookmarked but no other `onBookmark()` callback was set. It displays a modal dialog with the bookmark URL, along with a subtitle that is appropriate for the type of bookmarking used ("url" or "server").

**Usage**

```
showBookmarkUrlModal(url)
```

**Arguments**

- `url` A URL to show in the modal dialog.

---

**showModal**

*Show or remove a modal dialog*

**Description**

This causes a modal dialog to be displayed in the client browser, and is typically used with `modalDialog()`.

**Usage**

```
showModal(ui, session = getDefaultReactiveDomain())
removeModal(session = getDefaultReactiveDomain())
```

**Arguments**

- `ui` UI content to show in the modal.
- `session` The session object passed to function given to shinyServer.
showNotification

See Also

modalDialog() for examples.

Description

These functions show and remove notifications in a Shiny application.

Usage

showNotification(
  ui,
  action = NULL,
  duration = 5,
  closeButton = TRUE,
  id = NULL,
  type = c("default", "message", "warning", "error"),
  session = getDefaultReactiveDomain()
)

removeNotification(id, session = getDefaultReactiveDomain())

Arguments

ui
  Content of message.
action
  Message content that represents an action. For example, this could be a link that
  the user can click on. This is separate from ui so customized layouts can handle
  the main notification content separately from action content.
duration
  Number of seconds to display the message before it disappears. Use NULL to
  make the message not automatically disappear.
closeButton
  If TRUE, display a button which will make the notification disappear when clicked.
  If FALSE do not display.
id
  A unique identifier for the notification.
  id is optional for showNotification(): Shiny will automatically create one if
  needed. If you do supply it, Shiny will update an existing notification if it exists,
  otherwise it will create a new one.
  id is required for removeNotification().
type
  A string which controls the color of the notification. One of "default" (gray),
  "message" (blue), "warning" (yellow), or "error" (red).
session
  Session object to send notification to.

Value

An ID for the notification.
Examples

## Only run examples in interactive R sessions
if (interactive()) {
  # Show a message when button is clicked
  shinyApp(
    ui = fluidPage(
      actionButton("show", "Show")
    ),
    server = function(input, output) {
      observeEvent(input$show, {
        showNotification("Message text",
        action = a(href = "javascript:location.reload();", "Reload page")
      })
    })
  }
}

# App with show and remove buttons
shinyApp(
  ui = fluidPage(
    actionButton("show", "Show"),
    actionButton("remove", "Remove")
  ),
  server = function(input, output) {
    # A queue of notification IDs
    ids <- character(0)
    # A counter
    n <- 0
    observeEvent(input$show, {
      # Save the ID for removal later
      id <- showNotification(paste("Message", n), duration = NULL)
      ids <<- c(ids, id)
      n <<- n + 1
    })
    observeEvent(input$remove, {
      if (length(ids) > 0)
        removeNotification(ids[1])
      ids <<- ids[-1]
    })
  }
)

showTab  

Dynamically hide/show a tabPanel
Description
Dynamically hide or show a `tabPanel()` (or a `navbarMenu()`) from an existing `tabsetPanel()`, `navlistPanel()` or `navbarPage()`.

Usage
```
showTab(inputId, target, select = FALSE, session = getDefaultReactiveDomain())
hideTab(inputId, target, session = getDefaultReactiveDomain())
```

Arguments
- **inputId**: The id of the `tabsetPanel` (or `navlistPanel` or `navbarPage`) in which to find `target`.
- **target**: The value of the `tabPanel` to be hidden/shown. See Details if you want to hide/show an entire `navbarMenu` instead.
- **select**: Should `target` be selected upon being shown?
- **session**: The shiny session within which to call this function.

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

See Also
- `insertTab()`

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

  ui <- navbarPage("Nav bar page", id = "tabs",
    tabPanel("Home",
        actionButton("hideTab", "Hide 'Foo' tab"),
        actionButton("showTab", "Show 'Foo' tab"),
        actionButton("hideMenu", "Hide 'More' navbarMenu"),
        actionButton("showMenu", "Show 'More' navbarMenu"))
  ,
    tabPanel("Foo", "This is the foo tab"),
    tabPanel("Bar", "This is the bar tab"),
    navbarMenu("More",
        tabPanel("Table", "Table page"),
        tabPanel("About", "About page"),
        "------",
        "Even more!",
        tabPanel("Email", "Email page")
  )
  }
```
sidebarLayout

Layout a sidebar and main area

Description

Create a layout (sidebarLayout()) with a sidebar (sidebarPanel()) and main area (mainPanel()). The sidebar is displayed with a distinct background color and typically contains input controls. The main area occupies 2/3 of the horizontal width and typically contains outputs.

Usage

sidebarLayout(
  sidebarPanel,
  mainPanel,
  position = c("left", "right"),
  fluid = TRUE
)

sidebarPanel(..., width = 4)

mainPanel(..., width = 8)
Arguments

- **sidebarPanel**: The `sidebarPanel()` containing input controls.
- **mainPanel**: The `mainPanel()` containing outputs.
- **position**: The position of the sidebar relative to the main area ("left" or "right").
- **fluid**: TRUE to use fluid layout; FALSE to use fixed layout.
- **...**: Output elements to include in the sidebar/main panel.
- **width**: The width of the sidebar and main panel. By default, the sidebar takes up 1/3 of the width, and the main panel 2/3. The total width must be 12 or less.

See Also

Other layout functions: `fillPage()`, `fixedPage()`, `flowLayout()`, `fluidPage()`, `navbarPage()`, `splitLayout()`, `verticalLayout()`

Examples

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

  # Define UI
  ui <- fluidPage(

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

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

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

  )

  # Server logic
  server <- function(input, output) {
    output$distPlot <- renderPlot({ hist(rnorm(input$obs)) })
  }
}
```
# Complete app with UI and server components
shinyApp(ui, server)
}

## sizeGrowthRatio

**Create a sizing function that grows at a given ratio**

### Description

Returns a function which takes a two-element vector representing an input width and height, and returns a two-element vector of width and height. The possible widths are the base width times the growthRate to any integer power. For example, with a base width of 500 and growth rate of 1.25, the possible widths include 320, 400, 500, 625, 782, and so on, both smaller and larger. Sizes are rounded up to the next pixel. Heights are computed the same way as widths.

### Usage

```r
sizeGrowthRatio(width = 400, height = 400, growthRate = 1.2)
```

### Arguments

- **width, height**  
  Base width and height.
- **growthRate**  
  Growth rate multiplier.

### See Also

This is to be used with `renderCachedPlot()`.

### Examples

```r
f <- sizeGrowthRatio(500, 500, 1.25)
f(c(400, 400))
f(c(500, 500))
f(c(530, 550))
f(c(625, 700))
```
sliderInput  

*Slider Input Widget*

**Description**

Constructs a slider widget to select a number, date, or date-time from a range.

**Usage**

```r
sliderInput(
  inputId,  
  label,    
  min,      
  max,      
  value,    
  step = NULL, 
  round = FALSE, 
  ticks = TRUE, 
  animate = FALSE, 
  width = NULL, 
  sep = ",", 
  pre = NULL,  
  post = NULL, 
  timeFormat = NULL, 
  timezone = NULL, 
  dragRange = TRUE
)
```

```r
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, max**
  - The minimum and maximum values (inclusive) that can be selected.
- **value**
  - The initial value of the slider, either a number, a date (class Date), or a date-time (class POSIXxt). A length one vector will create a regular slider; a length two vector will create a double-ended range slider. Must lie between min and max.
- **step**
  - Specifies the interval between each selectable value on the slider. Either NULL, the default, which uses a heuristic to determine the step size or a single number. If the values are dates, step is in days; if the values are date-times, step is in seconds.
round
TRUE to round all values to the nearest integer; FALSE if no rounding is desired; or an integer to round to that number of digits (for example, 1 will round to the nearest 10, and -2 will round to the nearest .01). Any rounding will be applied after snapping to the nearest step.

ticks
FALSE to hide tick marks, TRUE to show them according to some simple heuristics.

animate
TRUE to show simple animation controls with default settings; FALSE not to; or a custom settings list, such as those created using animationOptions().

width
The width of the input, e.g. '400px', or '100%'; see validateCssUnit().

sep
Separator between thousands places in numbers.

pre
A prefix string to put in front of the value.

post
A suffix string to put after the value.

timeFormat
Only used if the values are Date or POSIXt objects. A time format string, to be passed to the Javascript strftime library. See https://github.com/samsonjs/strftime for more details. The allowed format specifications are very similar, but not identical, to those for R’s base::strftime() function. For Dates, the default is "%F" (like "2015-07-01"), and for POSIXt, the default is "%F %T" (like "2015-07-01 15:32:10").

timezone
Only used if the values are POSIXt objects. A string specifying the time zone offset for the displayed times, in the format "+HHMM" or "-HHMM". If NULL (the default), times will be displayed in the browser’s time zone. The value "+0000" will result in UTC time.

dragRange
This option is used only if it is a range slider (with two values). If TRUE (the default), the range can be dragged. In other words, the min and max can be dragged together. If FALSE, the range cannot be dragged.

interval
The interval, in milliseconds, between each animation step.

loop
TRUE to automatically restart the animation when it reaches the end.

playButton
Specifies the appearance of the play button. Valid values are a one-element character vector (for a simple text label), an HTML tag or list of tags (using tag() and friends), or raw HTML (using HTML()).

pauseButton
Similar to playButton, but for the pause button.

Server value
A number, date, or date-time (depending on the class of value), or in the case of slider range, a vector of two numbers/dates/date-times.

See Also
updateSliderInput()

Other input elements: actionButton(), checkboxGroupInput(), checkboxInput(), dateInput(), dateRangeInput(), fileInput(), numericInput(), passwordInput(), radioButtons(), selectInput(), submitButton(), textareaInput(), textInput(), varSelectInput()
## Examples

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

  ui <- fluidPage(
    sliderInput("obs", "Number of observations:",
        min = 0, max = 1000, value = 500
    ),
    plotOutput("distPlot")
  )

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

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

### snapshotExclude

Mark an output to be excluded from test snapshots

#### Description

Mark an output to be excluded from test snapshots

#### Usage

```r
snapshotExclude(x)
```

#### Arguments

- `x` A reactive which will be assigned to an output.

### snapshotPreprocessInput

Add a function for preprocessing an input before taking a test snapshot

#### Description

Add a function for preprocessing an input before taking a test snapshot
Usage

snapshotPreprocessInput(inputId, fun, session = getDefaultReactiveDomain())

Arguments

inputId Name of the input value.
fun A function that takes the input value and returns a modified value. The returned value will be used for the test snapshot.
session A Shiny session object.

Description

Add a function for preprocessing an output before taking a test snapshot

Usage

snapshotPreprocessOutput(x, fun)

Arguments

x A reactive which will be assigned to an output.
fun A function that takes the output value as an input and returns a modified value. The returned value will be used for the test snapshot.

Description

Lays out elements horizontally, dividing the available horizontal space into equal parts (by default).

Usage

splitLayout(...) cellWidths = NULL, cellArgs = list())

splitLayout Split layout
splitLayout

Arguments

... Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag.

cellWidths Character or numeric vector indicating the widths of the individual cells. Recycling will be used if needed. Character values will be interpreted as CSS lengths (see validateCssUnit()), numeric values as pixels.

cellArgs Any additional attributes that should be used for each cell of the layout.

See Also

Other layout functions: fillPage(), fixedPage(), flowLayout(), fluidPage(), navbarPage(), sidebarLayout(), verticalLayout()

Examples

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

  # Server code used for all examples
  server <- function(input, output) {
    output$plot1 <- renderPlot(plot(cars))
    output$plot2 <- renderPlot(plot(pressure))
    output$plot3 <- renderPlot(plot(AirPassengers))
  }

  # Equal sizing
  ui <- splitLayout(
    plotOutput("plot1"),
    plotOutput("plot2")
  )
  shinyApp(ui, server)

  # Custom widths
  ui <- splitLayout(cellWidths = c("25\%", "75\%"),
    plotOutput("plot1"),
    plotOutput("plot2")
  )
  shinyApp(ui, server)

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

Stop the currently running Shiny app

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

Usage
stopApp(returnValue = invisible())

Arguments
returnValue The value that should be returned from runApp().

submitButton
Create a submit button

Description
Create a submit button for an app. Apps that include a submit button do not automatically update their outputs when inputs change, rather they wait until the user explicitly clicks the submit button. The use of submitButton is generally discouraged in favor of the more versatile buttonText() (see details below).

Usage
submitButton(text = "Apply Changes", icon = NULL, width = NULL)

Arguments
text Button caption
icon Optional icon() to appear on the button
width The width of the button, e.g. '400px', or '100%'; see validateCssUnit().

Details
Submit buttons are unusual Shiny inputs, and we recommend using buttonText() instead of submitButton when you want to delay a reaction. See this article for more information (including a demo of how to "translate" code using a submitButton to code using a buttonText). In essence, the presence of a submit button stops all inputs from sending their values automatically to the server. This means, for instance, that if there are two submit buttons in the same app, clicking either one will cause all inputs in the app to send their values to the server. This is probably not what you’d want, which is why submit button are unwieldy for all but the simplest apps. There are other problems with submit buttons: for example, dynamically created submit buttons (for example, with renderUI() or insertUI()) will not work.
Value

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

See Also

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

Examples

```r
if (interactive()) {
  shinyApp(
    ui = basicPage(
      numericInput("num", label = "Make changes", value = 1),
      submitButton("Update View", icon("refresh")),
      helpText("When you click the button above, you should see",
        "the output below update to reflect the value you",
        "entered at the top:"),
      verbatimTextOutput("value")
    ),
    server = function(input, output) {
      # submit buttons do not have a value of their own,
      # they control when the app accesses values of other widgets.
      # input$num is the value of the number widget.
      output$value <- renderPrint({ input$num })
    }
  )
}
```

__tableOutput__  

**Table Output**

Description

The `tableOutput()` function creates a reactive table that is suitable for displaying small matrices and data frames. The columns are formatted with `xtable::xtable()`. See `renderDataTable()` for data frames that are too big to fit on a single page.

Usage

```r
tableOutput(outputId)

renderTable(
  expr,
  striped = FALSE,
```
Arguments

outputId    output variable to read the table from
expr        An expression that returns an R object that can be used with `xtable::xtable()`.
striped, hover, bordered
            Logicals: if TRUE, apply the corresponding Bootstrap table format to the output table.
spacing    The spacing between the rows of the table (xs stands for "extra small", s for "small", m for "medium" and l for "large").
width       Table width. Must be a valid CSS unit (like "100%", "400px", "auto") or a number, which will be coerced to a string and have "px" appended.
align       A string that specifies the column alignment. If equal to 'l', 'c' or 'r', then all columns will be, respectively, left-, center- or right-aligned. Otherwise, align must have the same number of characters as the resulting table (if rownames = TRUE, this will be equal to ncol()+1), with the i-th character specifying the alignment for the i-th column (besides 'l', 'c' and 'r', '?' is also permitted - '?' is a placeholder for that particular column, indicating that it should keep its default alignment). If NULL, then all numeric/integer columns (including the row names, if they are numbers) will be right-aligned and everything else will be left-aligned (align = '?' produces the same result).
rownames, colnames
            Logicals: include rownames? include colnames (column headers)?
digits      An integer specifying the number of decimal places for the numeric columns (this will not apply to columns with an integer class). If digits is set to a negative value, then the numeric columns will be displayed in scientific format with a precision of abs(digits) digits.
na          The string to use in the table cells whose values are missing (i.e. they either evaluate to NA or NaN).
...         Arguments to be passed through to `xtable::xtable()` and `xtable::print.xtable()`.
env         The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If expr is a quosure and quoted is TRUE, then env is ignored.
quoted
If it is TRUE, then the `quote()`ed value of `expr` will be used when `expr` is evaluated. If `expr` is a quosure and you would like to use its expression as a value for `expr`, then you must set quoted to TRUE.

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

Examples

```r
## 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)
    }
  )
}
```

---

tabPanel

*Create a tab panel*

Description

Create a tab panel

Usage

```r
tabPanel(title, ..., value = title, icon = NULL)
tabPanelBody(value, ..., 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()`.
tabsetPanel

Value

A tab that can be passed to `tabsetPanel()`

Functions

- `tabPanel()`: Create a tab panel that can be included within a `tabsetPanel()` or a `navbarPage()`.  
- `tabPanelBody()`: Create a tab panel that drops the title argument. This function should be used within `tabsetPanel(type = "hidden")`. See `tabsetPanel()` for example usage.

See Also

`tabsetPanel()`

Examples

```r
# 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

```r
tabsetPanel(
  ..., 
  id = NULL, 
  selected = NULL, 
  type = c("tabs", "pills", "hidden"), 
  header = NULL, 
  footer = NULL
)```
Arguments

... 

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  

"tabs" Standard tab look

"pills" Selected tabs use the background fill color

"hidden" Hides the selectable tabs. Use type = "hidden" in conjunction with tabPanelBody() and updateTabsetPanel() to control the active tab via other input controls. (See example below)

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

Value

A tabset that can be passed to mainPanel()

See Also

  tabPanel(), updateTabsetPanel(), insertTab(), showTab()

Examples

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

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      radioButtons("controller", "Controller", 1:3, 1)
    ),
    mainPanel(
      tabsetPanel(
        id = "hidden_tabs",
        # Hide the tab values.
        # Can only switch tabs by using 'updateTabsetPanel()'
        type = "hidden",
        tabPanelBody("panel1", "Panel 1 content"),
        tabPanelBody("panel2", "Panel 2 content"),
        tabPanelBody("panel3", "Panel 3 content")
      )
    )
  )
)
server <- function(input, output, session) {
  observeEvent(input$controller, {
    updateTabsetPanel(session, "hidden_tabs", selected = paste0("panel", input$controller))
  })
}

if (interactive()) {
  shinyApp(ui, server)
}

testServer

Reactive testing for Shiny server functions and modules

Description

A way to test the reactive interactions in Shiny applications. Reactive interactions are defined in the server function of applications and in modules.

Usage

testServer(app = NULL, expr, args = list(), session = MockShinySession$new())

Arguments

app A server function (i.e. a function with input, output, and session), or a module function (i.e. a function with first argument id that calls moduleServer()). You can also provide an app, a path an app, or anything that as.shiny.appobj() can handle.

expr Test code containing expectations. The objects from inside the server function environment will be made available in the environment of the test expression (this is done using a data mask with rlang::eval_tidy()). This includes the parameters of the server function (e.g. input, output, and session), along with any other values created inside of the server function.

args Additional arguments to pass to the module function. If app is a module, and no id argument is provided, one will be generated and supplied automatically.

session The MockShinySession object to use as the reactive domain. The same session object is used as the domain both during invocation of the server or module under test and during evaluation of expr.
Examples

# Testing a server function
server <- function(input, output, session) {
  x <- reactive(input$a * input$b)
}

testServer(server, {
  session$setInputs(a = 2, b = 3)
  stopifnot(x() == 6)
})

# Testing a module
myModuleServer <- function(id, multiplier = 2, prefix = "I am ") {
  moduleServer(id, function(input, output, session) {
    myreactive <- reactive({
      input$x * multiplier
    })
    output$txt <- renderText({
        paste0(prefix, myreactive())
    })
  })
}

testServer(myModuleServer, args = list(multiplier = 2), {
  session$setInputs(x = 1)
  session$setInputs(x = 2)
  session$setInputs(x = 2)
  # You're also free to use third-party
  # testing packages like testthat:
  # expect_equal(myreactive(), 2)
  stopifnot(myreactive() == 2)
  stopifnot(myreactive() == 2)
  stopifnot(output$txt == "I am 2")
  # Any additional arguments, below, are passed along to the module.
})

textAreaInput

Create a textarea input control

Description

Create a textarea input control for entry of unstructured text values.

Usage

textAreaInput(
  inputId,
label,  
value = "",  
width = NULL,  
height = NULL,  
cols = NULL,  
rows = NULL,  
placeholder = NULL,  
resize = NULL  
)

Arguments

inputId The input slot that will be used to access the value.
label Display label for the control, or NULL for no label.
value Initial value.
width The width of the input, e.g. '400px', or '100%'; see validateCssUnit().
height The height of the input, e.g. '400px', or '100%'; see validateCssUnit().
cols Value of the visible character columns of the input, e.g. 80. This argument will only take effect if there is not a CSS width rule defined for this element; such a rule could come from the width argument of this function or from a containing page layout such as fluidPage().
rows The value of the visible character rows of the input, e.g. 6. If the height argument is specified, height will take precedence in the browser's rendering.
placeholder A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option.
resize Which directions the textarea box can be resized. Can be one of "both", "none", "vertical", and "horizontal". The default, NULL, will use the client browser's default setting for resizing textareas.

Value

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

Server value

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

See Also

updateTextAreaInput()

Other input elements: actionButton(), checkboxGroupInput(), checkboxInput(), dateInput(), dateRangeInput(), fileInput(), numericInput(), passwordInput(), radioButtons(), selectInput(), sliderInput(), submitButton(), TextInput(), varSelectInput()
Examples
## Only run examples in interactive R sessions
if (interactive()) {

ui <- fluidPage(
  textAreaInput("caption", "Caption", "Data Summary", width = "1000px"),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({ input$caption })
}
shinyApp(ui, server)
}

textInput  Create a text input control

Description
Create an input control for entry of unstructured text values

Usage
textInput(inputId, label, value = "", width = NULL, placeholder = NULL)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputId</td>
<td>The input slot that will be used to access the value.</td>
</tr>
<tr>
<td>label</td>
<td>Display label for the control, or NULL for no label.</td>
</tr>
<tr>
<td>value</td>
<td>Initial value.</td>
</tr>
<tr>
<td>width</td>
<td>The width of the input, e.g. &quot;400px&quot;, or &quot;100%&quot;; see validateCssUnit().</td>
</tr>
<tr>
<td>placeholder</td>
<td>A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option.</td>
</tr>
</tbody>
</table>

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

Server value
A character string of the text input. The default value is "" unless value is provided.
See Also

updateTextInput()

Other input elements: 
actionButton(), checkboxGroupInput(), checkboxInput(), dateInput(), dateRangeInput(), fileInput(), numericInput(), passwordInput(), radioButtons(), selectInput(), sliderInput(), submitButton(), textareaInput(), varSelectInput()

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    textInput("caption", "Caption", "Data Summary"),
    verbatimTextOutput("value")
  )
  server <- function(input, output) {
    output$value <- renderText({ input$caption })
  }
  shinyApp(ui, server)
}
```

```r
textOutput

Create a text output element
```

Description

Render a reactive output variable as text within an application page. `textOutput()` is usually paired with `renderText()` and puts regular text in `<div>` or `<span>`; `verbatimTextOutput()` is usually paired with `renderPrint()` and provides fixed-width text in a `<pre>`.

Usage

```r
textOutput(outputId, container = if (inline) span else div, inline = FALSE)
verbatimTextOutput(outputId, placeholder = 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
- `placeholder`: if the output is empty or NULL, should an empty rectangle be displayed to serve as a placeholder? (does not affect behavior when the output is nonempty)

Details

In both functions, text is HTML-escaped prior to rendering.
Value

An output element for use in UI.

Examples

```r
### Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = basicPage(
      textInput("txt", "Enter the text to display below:"),
      textOutput("text"),
      verbatimTextOutput("verb")
    ),
    server = function(input, output) {
      output$text <- renderText({ input$txt })
      output$verb <- renderText({ input$txt })
    }
  )
}
```

## 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

```r
### Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    titlePanel("Hello Shiny!"
  )
}
```
updateActionButton

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

## Description

Change the label or icon of an action button on the client

## Usage

```r
updateActionButton(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  icon = NULL,
  disabled = NULL
)
```

```r
updateActionLink(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  icon = NULL
)
```

## Arguments

- **session**: The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`
- **inputId**: The id of the input object.
- **label**: The label to set for the input object.
- **icon**: An optional `icon()` to appear on the button.
- **disabled**: If TRUE, the button will not be clickable; if FALSE, it will be.

## Details

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

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

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

For `radioButtons()`, `checkboxGroupInput()` and `selectInput()`, the set of choices can be cleared by using `choices=character(0)`. Similarly, for these inputs, the selected item can be cleared by using `selected=character(0)`. 
See Also

actionButton()

Examples

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

ui <- fluidPage(
  actionButton("update", "Update other buttons and link"),
  br(),
  actionButton("goButton", "Go"),
  br(),
  actionButton("goButton2", "Go 2", icon = icon("area-chart")),
  br(),
  actionButton("goButton3", "Go 3"),
  br(),
  actionLink("goLink", "Go Link")
)

server <- function(input, output, session) {
  observe(
    req(input$update)
    # Updates goButton's label and icon
    updateActionButton(session, "goButton",
                       label = "New label",
                       icon = icon("calendar"))
    # Leaves goButton2's label unchanged and
    # removes its icon
    updateActionButton(session, "goButton2",
                        icon = character(0))
    # Leaves goButton3's icon, if it exists,
    # unchanged and changes its label
    updateActionButton(session, "goButton3",
                        label = "New label 3")
    # Updates goLink's label and icon
    updateActionButton(session, "goLink",
                        label = "New link label",
                        icon = icon("link"))
  )
}

shinyApp(ui, server)
}
updateCheckboxGroupInput

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

**Description**

Change the value of a checkbox group input on the client

**Usage**

```r
updateCheckboxGroupInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  choices = NULL,
  selected = NULL,
  inline = FALSE,
  choiceNames = NULL,
  choiceValues = NULL
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>The session object passed to function given to <code>shinyServer</code>. Default is <code>getDefaultReactiveDomain()</code></td>
</tr>
<tr>
<td>inputId</td>
<td>The id of the input object.</td>
</tr>
<tr>
<td>label</td>
<td>The label to set for the input object.</td>
</tr>
<tr>
<td>choices</td>
<td>List of values to show checkboxes for. If elements of the list are named then that name rather than the value is displayed to the user. If this argument is provided, then <code>choiceNames</code> and <code>choiceValues</code> must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings.</td>
</tr>
<tr>
<td>selected</td>
<td>The values that should be initially selected, if any.</td>
</tr>
<tr>
<td>inline</td>
<td>If TRUE, render the choices inline (i.e. horizontally)</td>
</tr>
<tr>
<td>choiceNames</td>
<td>List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, <code>choiceNames</code> and <code>choiceValues</code> must have the same length). If either of these arguments is provided, then the other <em>must</em> be provided and <code>choices</code> <em>must not</em> be provided. The advantage of using both of these over a named list for <code>choices</code> is that <code>choiceNames</code> allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples.</td>
</tr>
</tbody>
</table>
updateCheckboxGroupInput

Details

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

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

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

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

See Also

checkboxGroupInput()

Examples

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

ui <- fluidPage(
  p("The first checkbox group controls the second"),
  checkboxGroupInput("inCheckboxGroup", "Input checkbox",
            c("Item A", "Item B", "Item C")),
  checkboxGroupInput("inCheckboxGroup2", "Input checkbox 2",
            c("Item A", "Item B", "Item C"))
)

server <- function(input, output, session) {
  observe({
    x <- input$inCheckboxGroup
    # Can use character(0) to remove all choices
    if (is.null(x))
      x <- character(0)
    # Can also set the label and select items
    updateCheckboxGroupInput(session, "inCheckboxGroup2",
                        label = paste("Checkbox group label", length(x)),
                        choices = x,
                        selected = x
    )
  })
}
shinyApp(ui, server)
```
updateCheckboxInput  
*Change the value of a checkbox input on the client*

**Description**

Change the value of a checkbox input on the client

**Usage**

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

**Arguments**

- `session`: The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`.
- `inputId`: The id of the input object.
- `label`: The label to set for the input object.
- `value`: Initial value (TRUE or FALSE).

**Details**

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

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

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

For `radioButtons()`, `checkboxGroupInput()` and `selectInput()`, the set of choices can be cleared by using `choices=character(0)`. Similarly, for these inputs, the selected item can be cleared by using `selected=character(0)`.

**See Also**

`checkboxInput()`
Examples

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

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 1, 0, step = 1),
  checkboxInput("inCheckbox", "Input checkbox")
)

server <- function(input, output, session) {
  observe({
    # TRUE if input$controller is odd, FALSE if even.
    x_even <- input$controller %% 2 == 1

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

shinyApp(ui, server)
}
```

updateDateInput

Change the value of a date input on the client

Description

Change the value of a date input on the client

Usage

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

Arguments

- `session`: The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`.
- `inputId`: The id of the input object.
- `label`: The label to set for the input object.
- `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.
The minimum allowed date. Either a Date object, or a string in yyyy-mm-dd format.

max

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

Details

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

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

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

For radioButton(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

See Also

dateInput()

Examples

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

ui <- fluidPage(
  sliderInput("n", "Day of month", 1, 30, 10),
  dateInput("inDate", "Input date")
)

server <- function(input, output, session) {
  observe(
    date <- as.Date(paste0("2013-04-", input$n))
    updateDateInput(session, "inDate",
      label = paste("Date label", input$n),
      value = date,
      min   = date - 3,
      max   = date + 3
    )
  )
}

shinyApp(ui, server)
}
updateDateRangeInput  
Change the start and end values of a date range input on the client

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

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

Arguments

- session: The session object passed to function given to shinyServer. Default is getDefaultReactiveDomain().
- inputId: The id of the input object.
- label: The label to set for the input object.
- 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.

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

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

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

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).
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 = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  value = NULL,
  min = NULL,
  max = NULL,
  step = NULL
)
Arguments

- **session**: The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`.
- **inputId**: The id of the input object.
- **label**: The label to set for the input object.
- **value**: Initial value.
- **min**: Minimum allowed value.
- **max**: Maximum allowed value.
- **step**: Interval to use when stepping between min and max.

Details

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

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

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

For `radioButtons()`, `checkboxGroupInput()` and `selectInput()`, the set of choices can be cleared by using `choices=character(0)`. Similarly, for these inputs, the selected item can be cleared by using `selected=character(0)`.

See Also

- `numericInput()`

Examples

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

ui <- fluidPage(
  sliderInput("controller", "Controller", 0, 20, 10),
  numericInput("inNumber", "Input number", 0),
  numericInput("inNumber2", "Input number 2", 0)
)

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

  observeEvent(input$controller, {
    # We'll use the input$controller variable multiple times, so save it as x
    # for convenience.
    x <- input$controller
    updateNumericInput(session, "inNumber", value = x)
    updateNumericInput(session, "inNumber2",
    
    
```
updateQueryString

```r
  label = paste("Number label ", x),
  value = x, min = x-10, max = x+10, step = 5)
})

shinyApp(ui, server)
```

---

**updateQueryString**  
*Update URL in browser's location bar*

**Description**

This function updates the client browser's query string in the location bar. It typically is called from an observer. Note that this will not work in Internet Explorer 9 and below.

**Usage**

```r
updateQueryString(
  queryString,
  mode = c("replace", "push"),
  session = getDefaultReactiveDomain()
)
```

**Arguments**

- **queryString**: The new query string to show in the location bar.
- **mode**: When the query string is updated, should the current history entry be replaced (default), or should a new history entry be pushed onto the history stack? The former should only be used in a live bookmarking context. The latter is useful if you want to navigate between states using the browser's back and forward buttons. See Examples.
- **session**: A Shiny session object.

**Details**

For `mode = "push"`, only three updates are currently allowed:

1. the query string (format: ?param1=val1&param2=val2)
2. the hash (format: #hash)
3. both the query string and the hash (format: ?param1=val1&param2=val2#hash)

In other words, if `mode = "push"`, the `queryString` must start with either ? or with #.

A technical curiosity: under the hood, this function is calling the HTML5 history API (which is where the names for the mode argument come from). When `mode = "replace"`, the function called is `window.history.replaceState(null, null, queryString)`. When `mode = "push"`, the function called is `window.history.pushState(null, null, queryString)`. 
See Also

`enableBookmarking()`, `getQueryString()`

Examples

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

## App 1: Doing "live" bookmarking
## Update the browser's location bar every time an input changes.
## This should not be used with enableBookmarking("server"),
## because that would create a new saved state on disk every time
## the user changes an input.
enableBookmarking("url")
shinyApp(
  ui = function(req) {
    fluidPage(
      textInput("txt", "Text"),
      checkboxInput("chk", "Checkbox")
    ),
  },
  server = function(input, output, session) {
    observe(
      # Trigger this observer every time an input changes
      reactiveValuesToList(input)
      session$doBookmark()
    )
    onBookmarked(function(url) {
      updateQueryString(url)
    })
  }
)

## App 2: Printing the value of the query string
## (Use the back and forward buttons to see how the browser
## keeps a record of each state)
shinyApp(
  ui = fluidPage(
    textInput("txt", "Enter new query string"),
    helpText("Format: ?param1=val1&param2=val2"),
    actionButton("go", "Update"),
    hr(),
   verbatimTextOutput("query")
  ),
  server = function(input, output, session) {
    observeEvent(input$go, {
      updateQueryString(input$txt, mode = "push")
    })
    output$query <- renderText({
      query <- getQueryString()
      queryText <- paste(names(query), query,
        sep = "=", collapse=" ")
      queryText
    })
  }
)
```
updateRadioButtons

Change the value of a radio input on the client

Description

Change the value of a radio input on the client

Usage

```r
updateRadioButtons(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  choices = NULL,
  selected = NULL,
  inline = FALSE,
  choiceNames = NULL,
  choiceValues = NULL
)
```

Arguments

- `session`: The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`.
- `inputId`: The id of the input object.
- `label`: The label to set for the input object.
- `choices`: List of values to select from (if elements of the list are named then that name rather than the value is displayed to the user). If this argument is provided, then `choiceNames` and `choiceValues` must not be provided, and vice-versa. The values should be strings; other types (such as logicals and numbers) will be coerced to strings.
- `selected`: The initially selected value. If not specified, then it defaults to the first item in `choices`. To start with no items selected, use `character(0)`.
- `inline`: If `TRUE`, render the choices inline (i.e. horizontally).
- `choiceNames`, `choiceValues`: List of names and values, respectively, that are displayed to the user in the app and correspond to the each choice (for this reason, `choiceNames` and `choiceValues` must have the same length). If either of these arguments is provided, then the other **must** be provided and `choices` **must not** be provided. The advantage of using both of these over a named list for `choices` is that `choiceNames` allows any type of UI object to be passed through (tag objects, icons, HTML code, ...), instead of just simple text. See Examples.
Details

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

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

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

For `radioButtons()`, `checkboxGroupInput()` and `selectInput()`, the set of choices can be cleared by using `choices=character(0)`. Similarly, for these inputs, the selected item can be cleared by using `selected=character(0)`. 

See Also

`radioButtons()`

Examples

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

ui <- fluidPage(
  p("The first radio button group controls the second"),
  radioButtons("inRadioButtons", "Input radio buttons",
  c("Item A", "Item B", "Item C")),
  radioButtons("inRadioButtons2", "Input radio buttons 2",
  c("Item A", "Item B", "Item C"))
)

server <- function(input, output, session) {
  observe({
    x <- input$inRadioButtons

    # Can also set the label and select items
    updateRadioButtons(session, "inRadioButtons2",
      label = paste("radioButtons label", x),
      choices = x,
      selected = x
    )
  })
}

shinyApp(ui, server)
}
```
updateSelectInput  Change the value of a select input on the client

Description
Change the value of a select input on the client

Usage
updateSelectInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  choices = NULL,
  selected = NULL
)

updateSelectizeInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  choices = NULL,
  selected = NULL,
  options = list(),
  server = FALSE
)

updateVarSelectInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  data = NULL,
  selected = NULL
)

updateVarSelectizeInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  data = NULL,
  selected = NULL,
  options = list(),
  server = FALSE
)

Arguments

session The session object passed to function given to shinyServer. Default is getDefaultReactiveDomain()
inputId The id of the input object.
label The label to set for the input object.
choices List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user. It’s also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the group labels (leveraging the \texttt{<optgroup>} HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature.
selected The initially selected value (or multiple values if \texttt{multiple} = \texttt{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 \texttt{selectize.js(https://selectize.dev/docs/usage)} for possible options (character option values inside \texttt{base::I()} will be treated as literal JavaScript code; see \texttt{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 \texttt{selectize.js})
data A data frame. Used to retrieve the column names as choices for a \texttt{selectInput()}

Details

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

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

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

For \texttt{radioButtons()}, \texttt{checkboxGroupInput()} and \texttt{selectInput()}, the set of choices can be cleared by using \texttt{choices=character(0)}. Similarly, for these inputs, the selected item can be cleared by using \texttt{selected=character(0)}.

See Also

\texttt{selectInput()} \texttt{varSelectInput()}

Examples

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

ui <- fluidPage(
  p("The checkbox group controls the select input"),
  checkboxGroupInput("inCheckboxGroup", "Input checkbox",
    c("Item A", "Item B", "Item C")),
  selectInput("inSelect", "Select input",
    c("Item A", "Item B", "Item C"))
)
```
server <- function(input, output, session) {
  observe({
    x <- input$inCheckboxGroup

    # Can use character(0) to remove all choices
    if (is.null(x))
      x <- character(0)

    # Can also set the label and select items
    updateSelectInput(session, "inSelect",
                       label = paste("Select input label", length(x)),
                       choices = x,
                       selected = tail(x, 1)
      )
  })
}

shinyApp(ui, server)

updateSliderInput    Update Slider Input Widget

Description
Change the value of a slider input on the client.

Usage
updateSliderInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  value = NULL,
  min = NULL,
  max = NULL,
  step = NULL,
  timeFormat = NULL,
  timezone = NULL
)

Arguments

  session                The session object passed to function given to shinyServer. Default is getDefaultReactiveDomain()
  inputId                The id of the input object.
  label                  The label to set for the input object.
value  The initial value of the slider, either a number, a date (class Date), or a date-time (class POSIXt). A length one vector will create a regular slider; a length two vector will create a double-ended range slider. Must lie between min and max.

min, max  The minimum and maximum values (inclusive) that can be selected.

step  Specifies the interval between each selectable value on the slider. Either NULL, the default, which uses a heuristic to determine the step size or a single number. If the values are dates, step is in days; if the values are date-times, step is in seconds.

timeFormat  Only used if the values are Date or POSIXt objects. A time format string, to be passed to the Javascript strftime library. See https://github.com/samsonjs/strftime for more details. The allowed format specifications are very similar, but not identical, to those for R’s base::strftime() function. For Dates, the default is ”%F” (like "2015-07-01"), and for POSIXt, the default is ”%F %T” (like "2015-07-01 15:32:10").

timezone  Only used if the values are POSIXt objects. A string specifying the time zone offset for the displayed times, in the format ”+HHMM” or ”-HHMM”. If NULL (the default), times will be displayed in the browser’s time zone. The value ”+0000” will result in UTC time.

Details

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

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

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

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

See Also

sliderInput()

Examples

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

```r
```
updateTabsetPanel

```r
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**

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

```r
updateNavbarPage(
  session = getDefaultReactiveDomain(),
  inputId,
  selected = NULL
)
```

```r
updateNavlistPanel(
  session = getDefaultReactiveDomain(),
  inputId,
  selected = NULL
)
```

**Arguments**

- `session` The session object passed to function given to `shinyServer`. Default is `getDefaultReactiveDomain()`.
- `inputId` The id of the tabsetPanel, navlistPanel, or navbarPage object.
updateTextAreaInput

Description

Change the value of a textarea input on the client

Usage

updateTextAreaInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  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.

See Also

tabsetPanel(), navlistPanel(), navbarPage()

Examples

## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(sidebarLayout(
    sidebarPanel(
      sliderInput("controller", "Controller", 1, 3, 1)
    ),
    mainPanel(
      tabsetPanel(id = "inTabset",
        tabPanel(title = "Panel 1", value = "panel1", "Panel 1 content"),
        tabPanel(title = "Panel 2", value = "panel2", "Panel 2 content"),
        tabPanel(title = "Panel 3", value = "panel3", "Panel 3 content")
      )
    )
  )
  )
  server <- function(input, output, session) {
    observeEvent(input$controller, {
      updateTabsetPanel(session, "inTabset",
        selected = paste0("panel", input$controller)
      )
    })
  }
  shinyApp(ui, server)
}
updateTextAreaInput

value = NULL,
placeholder = NULL
)

Arguments

session The session object passed to function given to shinyServer. Default is getDefaultReactiveDomain().
inputId The id of the input object.
label The label to set for the input object.
value Initial value.
placeholder A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option.

Details

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

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

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

For radioButtons(), checkboxGroupInput() and selectInput(), the set of choices can be cleared by using choices=character(0). Similarly, for these inputs, the selected item can be cleared by using selected=character(0).

See Also
textAreaInput()

Examples

## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    sliderInput("controller", "Controller", 0, 20, 10),
    textAreaInput("inText", "Input textarea"),
    textAreaInput("inText2", "Input textarea 2")
  )

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

      # This will change the value of input$inText, based on x
      updateTextAreaInput(session, "inText", value = paste("New text", x))
    )
  }
}
# Can also set the label, this time for input$inText2
df <- df %>%
  mutate(
inText2 = ifelse(inText2 == "Old value", "New value", inText2))
}
}

shinyApp(ui, server)
}

updateTextInput

Change the value of a text input on the client

Description

Change the value of a text input on the client

Usage

updateTextInput(
  session = getDefaultReactiveDomain(),
  inputId,
  label = NULL,
  value = NULL,
  placeholder = NULL
)

Arguments

session The session object passed to function given to shinyServer. Default is getDefaultReactiveDomain()
inputId The id of the input object.
label The label to set for the input object.
value Initial value.
placeholder A character string giving the user a hint as to what can be entered into the control. Internet Explorer 8 and 9 do not support this option.

Details

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

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

Any arguments with NULL values will be ignored; they will not result in any changes to the input object on the client.
For **radioButtons()**, **checkboxGroupInput()** and **selectInput()**, the set of choices can be cleared by using `choices=character(0)`. Similarly, for these inputs, the selected item can be cleared by using `selected=character(0)`.

**See Also**

textInput()

**Examples**

```r
## Only run examples in interactive R sessions
if (interactive()) {
  ui <- fluidPage(
    sliderInput("controller", "Controller", 0, 20, 10),
    textInput("inText", "Input text"),
    textInput("inText2", "Input text 2")
  )

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

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

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

  shinyApp(ui, server)
}
```

**urlModal**

*Generate a modal dialog that displays a URL*

**Description**

The modal dialog generated by `urlModal` will display the URL in a textarea input, and the URL text will be selected so that it can be easily copied. The result from `urlModal` should be passed to the `showModal()` function to display it in the browser.

**Usage**

```r
urlModal(url, title = "Bookmarked application link", subtitle = NULL)
```
Arguments

url A URL to display in the dialog box.
title A title for the dialog box.
subtitle Text to display underneath URL.

Description

Busy indicators provide a visual cue to users when the server is busy calculating outputs or otherwise performing tasks (e.g., producing downloads). When enabled, a spinner is shown on each calculating/recalculating output, and a pulsing banner is shown at the top of the page when the app is otherwise busy. Busy indication is enabled by default for UI created with `bslib`, but must be enabled otherwise. To enable/disable, include the result of this function in anywhere in the app’s UI.

Usage

```r
useBusyIndicators(..., spinners = TRUE, pulse = TRUE, fade = TRUE)
```

Arguments

... Currently ignored.
spinners Whether to show a spinner on each calculating/recalculating output.
pulse Whether to show a pulsing banner at the top of the page when the app is busy.
fade Whether to fade recalculating outputs. A value of FALSE is equivalent to `busyIndicatorOptions(fade_opacity=1)`.

Details

When both `spinners` and `pulse` are set to TRUE, the pulse is automatically disabled when spinner(s) are active. When both `spinners` and `pulse` are set to FALSE, no busy indication is shown (other than the graying out of recalculating outputs).

See Also

`busyIndicatorOptions()` for customizing the appearance of the busy indicators.

Examples

```r
library(bslib)
ui <- page_fillable(
  useBusyIndicators(),
  card(
```
server <- function(input, output) {
  output$p <- renderPlot({
    input$simulate
    Sys.sleep(4)
    plot(x = rnorm(100), y = rnorm(100))
  })
}

shinyApp(ui, server)

validate(...) Validate input values and other conditions

Description

validate() provides convenient mechanism for validating that an output has all the inputs necessary for successful rendering. It takes any number of (unnamed) arguments, each representing a condition to test. If any of condition fails (i.e. is not "truthy"), a special type of error is signaled to stop execution. If this error is not handled by application-specific code, it is displayed to the user by Shiny.

If you use validate() in a reactive() validation failures will automatically propagate to outputs that use the reactive.

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.

need()  
An easy way to provide arguments to validate() is to use need(), which takes an expression and a string. If the expression is not "truthy" then the string will be used as the error message.

If "truthiness" is flexible for your use case, you'll need to explicitly generate a logical values. For example, if you want allow NA but not NULL, you can !is.null(input$foo).

If you need validation logic that differs significantly from need(), you can create your own validation test functions. A passing test should return NULL. A failing test should return either a string providing the error to display to the user, or if the failure should happen silently, FALSE.

Alternatively you can use validate() within an if statement, which is particularly useful for more complex conditions:

```r
if (input$x < 0 && input$choice == "positive") {
  validate("If choice is positive then x must be greater than 0")
}
```

Examples

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

  ui <- fluidPage(
    checkboxGroupInput('in1', 'Check some letters', choices = head(LETTERS)),
    selectizeInput('in2', 'Select a state', choices = c('', state.name)),
    plotOutput('plot')
  )

  server <- function(input, output) {
    output$plot <- renderPlot({
      validate(  
        need(input$in1, 'Check at least one letter!'),
        need(input$in2 != '', 'Please choose a state.'),
      )
      plot(1:10, main = paste(c(input$in1, input$in2), collapse = ', '))
    })
  }

  shinyApp(ui, server)
}
```
varSelectInput

Select variables from a data frame

Description

Create a select list that can be used to choose a single or multiple items from the column names of a data frame.

Usage

```
varSelectInput(
    inputId, label, data, selected = NULL, multiple = FALSE, selectize = TRUE, width = NULL, size = NULL
)
```

```
varSelectizeInput(inputId, ..., options = NULL, width = NULL)
```

Arguments

- `inputId` The input slot that will be used to access the value.
- `label` Display label for the control, or NULL for no label.
- `data` A data frame. Used to retrieve the column names as choices for a selectInput()
- `selected` The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
- `multiple` Is selection of multiple items allowed?
- `selectize` Whether to use selectize.js or not.
- `width` The width of the input, e.g. '400px', or '100%'; see validateCssUnit().
- `size` Number of items to show in the selection box; a larger number will result in a taller box. Not compatible with selectize=TRUE. Normally, when multiple=FALSE, a select input will be a drop-down list, but when size is set, it will be a box instead.
- `...` Arguments passed to varSelectInput().
- `options` A list of options. See the documentation of selectize.js(https://selectize.dev/docs/usage) for possible options (character option values inside base::I() will be treated as literal JavaScript code; see renderDataTable() for details).
Details

By default, varSelectInput() and selectizeInput() use the JavaScript library `selectize.js` (https://selectize.dev/) to instead of the basic select input element. To use the standard HTML select input element, use selectInput() with selectize=FALSE.

Value

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

Server value

The resulting server input value will be returned as:

- A symbol if multiple = FALSE. The input value should be used with rlang's `rlang::!!()`.
  For example, `ggplot2::aes(!!input$variable)`.
- A list of symbols if multiple = TRUE. The input value should be used with rlang's `rlang::!!!()`
  to expand the symbol list as individual arguments. For example, `dplyr::select(mtcars, 
  !!!input$variables) which is equivalent to dplyr::select(mtcars, !!input$variables[[1]], 
  !!input$variables[[2]], ..., !!input$variables[[length(input$variables)])`.

Note

The variable selectize input created from varSelectizeInput() allows deletion of the selected option even in a single select input, which will return an empty string as its value. This is the default behavior of selectize.js. However, the selectize input created from selectInput(..., selectize = TRUE) will ignore the empty string value when it is a single choice input and the empty string is not in the choices argument. This is to keep compatibility with selectInput(..., selectize = FALSE).

See Also

`updateSelectInput()`

Other input elements: `actionButton()`, `checkboxGroupInput()`, `checkboxInput()`, `dateInput()`, `dateRangeInput()`, `fileInput()`, `numericInput()`, `passwordInput()`, `radioButtons()`, `selectInput()`, `sliderInput()`, `submitButton()`, `textareaInput()`, `textInput()`

Examples

```r
## Only run examples in interactive R sessions
if (interactive()) {
  library(ggplot2)

  # single selection
  shinyApp(
    ui = fluidPage(
      varSelectInput("variable","Variable:", mtcars),
      plotOutput("data"")
    ),
    server = function(input, output) {
```
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 its own line in the UI)

Usage

verticalLayout(..., fluid = TRUE)

Arguments

... Elements to include within the container
fluid TRUE to use fluid layout; FALSE to use fixed layout.

See Also

Other layout functions: fillPage(), fixedPage(), flowLayout(), fluidPage(), navbarPage(), sidebarLayout(), splitLayout()
## Examples

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

  ui <- fluidPage(
    verticalLayout(
      a(href="http://example.com/link1", "Link One"),
      a(href="http://example.com/link2", "Link Two"),
      a(href="http://example.com/link3", "Link Three")
    )
  )
  shinyApp(ui, server = function(input, output) { })
}
```

---

### Viewer options

**Description**

Use these functions to control where the gadget is displayed in RStudio (or other R environments that emulate RStudio's viewer/dialog APIs). If viewer APIs are not available in the current R environment, then the gadget will be displayed in the system's default web browser (see `utils::browseURL()`).

**Usage**

- `paneViewer(minHeight = NULL)`
- `dialogViewer(dialogName, width = 600, height = 600)`
- `browserViewer(browser = getOption("browser"))`

**Arguments**

- `minHeight` The minimum height (in pixels) desired to show the gadget in the viewer pane. If a positive number, resize the pane if necessary to show at least that many pixels. If NULL, use the existing viewer pane size. If "maximize", use the maximum available vertical space.
- `dialogName` The window title to display for the dialog.
- `width, height` The desired dialog width/height, in pixels.
- `browser` See `utils::browseURL()`.

**Value**

A function that takes a single `url` parameter, suitable for passing as the `viewer` argument of `runGadget()`.
wellPanel

Create a well panel

Description

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

Usage

`wellPanel(...)`

Arguments

... UI elements to include inside the panel.

Value

The newly created panel.

withMathJax

Load the MathJax library and typeset math expressions

Description

This function adds MathJax to the page and typeset the math expressions (if found) in the content. ... It only needs to be called once in an app unless the content is rendered after the page is loaded, e.g. via `renderUI()`, in which case we have to call it explicitly every time we write math expressions to the output.

Usage

`withMathJax(...)`

Arguments

... any HTML elements to apply MathJax to

Examples

```r
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**

```r
withProgress(
  expr,
  min = 0,
  max = 1,
  value = min + (max - min) * 0.1,
  message = NULL,
  detail = NULL,
  style = getShinyOption("progress.style", default = "notification"),
  session = getDefaultReactiveDomain(),
  env = parent.frame(),
  quoted = FALSE
)

setProgress(
  value = NULL,
  message = NULL,
  detail = NULL,
  session = getDefaultReactiveDomain()
)

incProgress(
  amount = 0.1,
  message = NULL,
  detail = NULL,
  session = getDefaultReactiveDomain()
)
```

**Arguments**

- **expr**
  - The work to be done. This expression should contain calls to `setProgress()` or `incProgress()`.

- **min**
  - The value that represents the starting point of the progress bar. Must be less than `max`. Default is 0.

- **max**
  - The value that represents the end of the progress bar. Must be greater than `min`. Default is 1.

- **value**
  - Single-element numeric vector; the value at which to set the progress bar, relative to `min` and `max`. 
message
A single-element character vector; the message to be displayed to the user, or
NULL to hide the current message (if any).
detail
A single-element character vector; the detail message to be displayed to the user,
or NULL to hide the current detail message (if any). The detail message will be
shown with a de-emphasized appearance relative to message.
style
Progress display style. If "notification" (the default), the progress indicator
will show using Shiny’s notification API. If "old", use the same HTML and
CSS used in Shiny 0.13.2 and below (this is for backward-compatibility).
session
The Shiny session object, as provided by shinyServer to the server function.
The default is to automatically find the session by using the current reactive
domain.
env
The environment in which expr should be evaluated.
quoted
Whether expr is a quoted expression (this is not common).
amount
For incProgress, the amount to increment the status bar. Default is 0.1.

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

Use withProgress to wrap the scope of your work; doing so will cause a new progress panel to
be created, and it will be displayed the first time incProgress or setProgress are called. When
withProgress exits, the corresponding progress panel will be removed.

The incProgress function increments the status bar by a specified amount, whereas the setProgress
function sets it to a specific value, and can also set the text displayed.

Generally, withProgress/incProgress/setProgress should be sufficient; the exception is if the
work to be done is asynchronous (this is not common) or otherwise cannot be encapsulated by a
single scope. In that case, you can use the Progress reference class.

As of version 0.14, the progress indicators use Shiny’s new notification API. If you want to use the
old styling (for example, you may have used customized CSS), you can use style="old" each time
you call withProgress(). If you don’t want to set the style each time withProgress is called, you
can instead call shinyOptions(progress.style="old") just once, inside the server function.

Value
The result of expr.

See Also
Progress()

Examples
## Only run examples in interactive R sessions
if (interactive()) {
  options(device.ask.default = FALSE)
ui <- fluidPage(
  plotOutput("plot")
)

server <- function(input, output) {
  output$plot <- renderPlot({
    withProgress(message = 'Calculation in progress',
                 detail = 'This may take a while...',
                 value = 0, {
      for (i in 1:15) {
        incProgress(1/15)
        Sys.sleep(0.25)
      }
    })
    plot(cars)
  })
}

shinyApp(ui, server)
Index

* datasets
  NS, 115

* input elements
  actionButton, 9
  checkboxGroupInput, 32
  checkboxInput, 34
  dateInput, 43
  dateRangeInput, 45
  fileInput, 66
  numericInput, 116
  passwordInput, 133
  radioButton, 143
  selectInput, 182
  sliderInput, 200
  submitButton, 205
  textAreaInput, 212
  TextInput, 214
  varSelectInput, 243

* layout functions
  fillPage, 68
  fixedPage, 71
  flowLayout, 73
  fluidPage, 73
  navbarPage, 111
  sidebarLayout, 197
  splitLayout, 203
  verticalLayout, 245

absolutePanel, 8
actionButton, 9, 22, 33, 34, 44, 47, 67, 116, 134, 144, 184, 201, 206, 213, 215, 244
actionButton(), 20–22, 95, 121, 122, 205, 218
actionLink (actionButton), 9
addResourcePath, 11
addResourcePath(), 42
animationOptions (sliderInput), 200
animationOptions(), 201
appendTab (insertTab), 86
applyInputHandlers(), 157
as.shiny.appobj(), 211
base::.Random.seed(), 171
base::as.list(), 154
base::I(), 183, 232, 243
base::local(), 93
base::logical(), 95
base::strptime(), 201, 234
base::Sys.time(), 151
basicPage (bootstrapPage), 24
bindCache, 12
bindCache(), 13, 19, 21, 40, 98, 119, 159, 160
bindEvent, 19
bindEvent(), 13, 14, 17, 65, 119
bookmarkButton, 22
bootstrapLib, 23
bootstrapLib(), 188
bootstrapPage, 24
bootstrapPage(), 24
browserViewer (viewer), 246
brushedPoints, 25
brushOpts, 27
brushOpts(), 35, 135, 136
bslib::bs_theme(), 24, 68, 71, 74, 107, 112
busyIndicatorOptions, 28
busyIndicatorOptions(), 240
cachem::cache_disk(), 12, 14, 15, 159, 160
cachem::cache_mem(), 15, 83, 160
Cairo::CairoPNG(), 140
callModule, 31
callModule(), 31, 110
cat(), 167
checkboxGroupInput, 10, 32, 34, 44, 47, 67, 116, 134, 144, 184, 201, 206, 213, 215, 217, 220, 221, 223, 224, 226, 230, 232, 234, 237, 239, 244
checkboxGroupInput(), 34, 220
INDEX

icon(), 10, 22, 56, 107, 205, 217
imageOutput (plotOutput), 134
imageOutput(), 27, 35, 164
in_devnode (devmode), 50
incProgress (withProgress), 248
incProgress(), 248
inputPanel, 86
insertTab, 86
insertTab(), 113, 114, 196, 210
insertUI, 89
insertUI(), 205
installExprFunction
  (createRenderFunction), 38
invalidateLater, 91
invalidateLater(), 151
invisible, 168
is.reactive (reactive), 146
is.reactivevalues, 93
is.reactivevalues(), 154
isolate, 93
isolate(), 20, 65, 82, 99, 120, 121, 154
isRunning, 94
isTruthy, 95

list.files, 96
loadSupport, 96

mainPanel (sidebarLayout), 197
mainPanel(), 210
markdown, 96
markRenderFunction, 98
markRenderFunction(), 17, 40
maskReactiveContext, 99
MockShinySession, 100, 211
modalButton (modalDialog), 106
modalDialog, 106
modalDialog(), 193, 194
moduleServer, 109
moduleServer(), 31, 192, 211

namespace(), 37
navbarMenu (navbarPage), 111
navbarMenu(), 85–87, 196
navbarPage, 69, 72–74, 111, 198, 204, 245
navbarPage(), 24, 70, 86, 87, 196, 208, 209, 236
navlistPanel, 113
navlistPanel(), 86, 196, 236
nearPoints (brushedPoints), 25
need (validate), 241
need(), 10
NS, 115
ns. sep (NS), 115
numericInput, 10, 33, 34, 44, 47, 67, 116,
  134, 144, 184, 201, 206, 213, 215,
  217, 220, 221, 223, 224, 226, 230,
  232, 234, 237, 238, 244
numericInput(), 226

observe, 117
observe(), 19, 20, 55, 121, 173
observeEvent, 119
observeEvent(), 10, 19, 20, 65, 173
observers, 48
onBookmark, 124
onBookmark(), 59, 193
onBookmarked (onBookmark), 124
onBookmarked(), 59
onFlush, 128
onFlushed (onFlush), 128
onReactiveDomainEnded (domains), 54
onRestore (onBookmark), 124
onRestore(), 59
onRestored (onBookmark), 124
onRestored(), 59
onSessionEnded (onFlush), 128
onSessionEnded(), 130
onStop, 130
onStop(), 129
onUnhandledError (onFlush), 128
options(), 50, 51, 80
outputOptionsMenu, 131

paneViewer (viewer), 246
parseQueryString, 132
parseQueryString(), 186
passwordInput, 10, 33, 34, 44, 47, 67, 116,
  133, 144, 184, 201, 206, 213, 215,
  244
plotOutput, 134
plotOutput(), 27, 35, 73, 160, 166, 167
plotPNG, 139
plotPNG(), 82, 159, 164, 166, 167
prependTab (insertTab), 86
print(), 167
Progress, 140
Progress(), 249
promises::as.promise(), 64
promises::promise(), 15
quote(), 117, 120, 121, 146, 164, 166, 168, 170, 208
quoToFunction(createRenderFunction), 38
radioButtons, 10, 33, 34, 44, 47, 67, 116, 134, 143, 184, 201, 206, 213, 215, 217, 220, 221, 223, 224, 226, 230, 232, 234, 237, 239, 244
radioButtons(), 230
ragg::agg_png(), 139
reactive(), 146
reactive domain, 211
reactive expression, 20, 121
Reactive expressions, 151
reactive(), 12, 13, 19, 20, 55, 121, 173, 241
reactiveReader, 147
reactiveReader(), 150
reactivePoll(), 149
reactivePoll(), 148
reactiveTimer, 150
reactiveTimer(), 92
reactiveVal, 152
reactiveVal(), 76
reactiveValues, 153
reactiveValues(), 76, 93, 152, 186
reactiveValuesToList, 154
reactlog, 155
reactlog(), 152
reactlog::reactlog_show, 156
reactlogAddMark(reactlog), 155
reactlogReset(reactlog), 155
reactlogShow(reactlog), 155
reactlogShow(), 82
register_devmode_option(devmode), 50
registerInputHandler, 157
registerInputHandler(), 158
registerThemeDependency, 188
removeInputHandler(), 158
removeInputHandler(), 157
removeModal(showModal), 193
removeModal(), 107
removeNotification(showNotification), 194
removeResourcePath(addResourcePath), 11
removeTab(insertTab), 86
removeUI(insertUI), 89
renderCachedPlot(), 17, 83, 167, 199
renderDataTable(), 183, 206, 232, 243
renderImage, 163
renderImage(), 28, 134, 136
renderPlot, 40, 98, 165
renderPlot(), 19, 119, 134, 136, 159, 160
renderPrint, 167
renderPrint(), 215
renderTable, 206
renderTable(), 12
renderText, 167
renderUI, 12, 16, 19, 119, 215
restoredInput, 169
removeResourcePath(addResourcePath), 11
restoreInput(), 89, 205, 247
repeatable, 171
req, 171
req(), 10, 76
req(FALSE), 65
resourcePaths (addResourcePath), 11
runApp, 174
runApp(), 59, 81–83, 130, 156, 177, 180, 191, 205
runExample, 176
runGadget, 177
runGadget(), 246
runGist (runUrl), 179
runGitHub (runUrl), 179
runTests, 178
runTests(), 192
runUrl, 179
safeError, 181
setInput(), 10, 33, 34, 44, 47, 67, 116, 134, 144, 182, 201, 206, 213, 215, 217, 220, 221, 223, 224, 226, 230, 232, 234, 237, 239, 244
selectInput(), 232, 243
selectizeInput (selectInput), 182
setBookmarkExclude, 188
setBookmarkExclude(), 23
setProgress (withProgress), 248
setProgress(), 141, 248
setSerializer, 189
shiny (shiny-package), 6
shiny-options, 7
shiny-options (getShinyOption), 80
shiny-package, 6
shinyApp, 189
shinyApp(), 59, 174, 177
shinyAppDir (shinyApp), 189
shinyAppFile (shinyApp), 189
shinyAppTemplate, 191
shinyAppTemplate(), 179
shinyDeprecated(), 81
shinyOptions (getShinyOption), 80
shinyOptions(), 14
shinyOptions(progress.style=old), 141, 249
shinyUI, 25, 72, 74, 113
showBookmarkUrlModal, 193
showModal, 193
showModal(), 239
showNotification, 194
showTab(), 87, 113, 114, 210
sidebarLayout, 69, 72–74, 113, 197, 204, 245
sidebarLayout(), 25, 70, 74
sidebarPanel (sidebarLayout), 197
singleton(), 12
sizeGrowthRatio, 199
sizeGrowthRatio(), 17, 159
sliderInput, 10, 22, 33, 34, 44, 47, 67, 116, 134, 144, 184, 200, 206, 213, 215, 244
sliderInput(), 234
snapshotExclude, 202
snapshotPreprocessInput, 202
snapshotPreprocessOutput, 203
splitLayout, 69, 72–74, 113, 198, 203, 245
stacktrace(), 40, 118, 146
stderr(), 51
stopApp, 205
submitButton, 10, 33, 34, 44, 47, 67, 116, 134, 144, 184, 201, 205, 213, 215, 244
tableOutput, 206
tableOutput(), 208
tabPanel, 208
tabPanel(), 85, 86, 111–114, 196, 209, 210
tabPanelBody (tabPanel), 208
tabPanelBody(), 210
tabsetPanel, 209
tabsetPanel(), 86, 113, 196, 209, 236
tag(), 201
tagList(), 70
testServer, 211
testServer(), 31, 100, 110
textAreaInput, 10, 33, 34, 44, 47, 67, 116, 134, 144, 184, 201, 206, 212, 215, 244
textAreaInput(), 237
textInput, 10, 33, 34, 44, 47, 67, 116, 134, 144, 184, 201, 206, 213, 214, 244
textInput(), 173, 239
textOutput, 215
textOutput(), 167, 168
throttle (debounce), 48
titlePanel, 216
titlePanel(), 74
uiOutput (htmlOutput), 84
uiOutput(), 170
updateActionButton, 217
updateActionButton(), 10
updateActionLink (updateActionButton), 217
updateCheckboxGroupInput, 219
updateCheckboxGroupInput(), 33
updateCheckboxInput, 221
updateCheckboxInput(), 34
updateDateInput, 222
updateDateInput(), 44
updateDateRangeInput, 224
updateDateRangeInput(), 47
updateNavlistPanel (updateTabsetPanel), 235
updateNavbarPage (updateTabsetPanel), 235
updateNavlistPanel(), 113
updateNavlistPanel (updateTabsetPanel), 235
updateNavlistPanel(), 114
updateNumericInput, 225
updateNumericInput(), 116
updateQueryString, 227
updateQueryString(), 59, 79
updateRadioButtons, 229
updateRadioButtons(), 144
updateSelectInput, 231
updateSelectInput(), 184, 244
updateSelectizeInput
  (updateSelectInput), 231
updateSliderInput, 233
updateSliderInput(), 201
updateTabsetPanel, 235
updateTabsetPanel(), 210
updateTextAreaInput, 236
updateTextAreaInput(), 213
updateTextInput, 238
updateTextInput(), 134, 187, 215
updateVarSelectInput
  (updateSelectInput), 231
updateVarSelectizeInput
  (updateSelectInput), 231
urlModal, 239
urlModal(), 193
useBusyIndicators, 240
useBusyIndicators(), 30
utils::browseURL(), 246
validate, 241
validateCssUnit(), 10, 32, 34, 44, 46, 66,
  116, 133, 144, 183, 201, 204, 205,
  213, 214, 243
validation, 21, 122
varSelectInput, 10, 33, 34, 44, 47, 67, 116,
  134, 144, 184, 201, 206, 213, 215,
  243
varSelectInput(), 184, 232
varSelectizeInput (varSelectInput), 243
verbatimTextOutput (textOutput), 215
verbatimTextOutput(), 167, 168
verticalLayout, 69, 72–74, 113, 198, 204,
  245
viewer, 246
viewer(), 177
wellPanel, 247
with_devmode (devmode), 50
withMathJax, 247
withMathJax(), 81
withProgress, 248
withProgress(), 141, 142
withReactiveDomain (domains), 54
xtable::print.xtable(), 207
xtable::xtable(), 206, 207