Package 'shinyGizmo'

July 23, 2025

Type Package

Title Custom Components for Shiny Applications

Version 0.4.2

Maintainer Krystian Igras <krystian8207@gmail.com>

Description Provides useful UI components and input widgets for 'Shiny' applications. The offered components allow to apply non-standard operations and view to your 'Shiny' application, but also help to overcome common performance issues.

License MIT + file LICENSE

Imports glue, rlang, htmltools, htmlwidgets, magrittr, purrr, shiny (>= 1.5.0), shinyWidgets (>= 0.7.0)

Encoding UTF-8

RoxygenNote 7.2.3

NeedsCompilation no

Author Krystian Igras [cre, aut],
Adam Forys [ctb],
Stéphane Laurent [ctb],
Craig Dennis [cph] (jQuery animateCSS),
Daniel Eden [cph] (Animate.css)

Repository CRAN

Date/Publication 2023-03-01 09:40:03 UTC

Contents

`shinyGizmo-package` *Useful Components For Shiny Applications*

Description

Useful Components For Shiny Applications

`.cssEffects` *Supported animation effects*

Description

Can be used as ‘effectShow‘ and ‘effectHide‘ arguments of animateVisibility, or ‘effect‘ of runAnimation.

Usage

```
.cssEffects
```

Format

An object of class `character` of length 97.

`accordion` *Create simple accordion*

Description

Created component provides basic accordion functionality - enroll/collapse behavior with only necessary styling (enrolled state icon). In order to provide custom styling for accordion items configure its header and content accordingly while constructing the item (see accordionItem).

Usage

```
accordion(id, ..., class = "")
```

Arguments

`id`	Id of the accordion component.
`...`	Accordion items created with accordionItem.
`class`	Extra class added to accordion container.

Value

A 'shiny.tag' object defining html structure of accordion container.

Examples

```
# Basic construction with simple header and content
if (interactive()) {
  library(shiny)
  ui <- fluidPage(
    actionButton("new", "New"),
    accordion(
      "acc",
      accordionItem("first", "Hello", "There", active = TRUE),
      accordionItem("second", "General", "Kenobi")
    )
  )

  server <- function(input, output, session) {
    observeEvent(input$new, {
    addAccordionItem("acc", accordionItem(sample(letters, 1), "New", "Accordion", active = TRUE))
    })
  }

  shinyApp(ui, server)
}
```

`accordionEnrollOnClick` *Define Enroll/Collapse trigger*

Description

The function is useful if you want to override standard behavior for activating accordion item. By default accordion item is activated when its header is clicked.

In order to point the trigger to a different object (included in item's header or content) attach 'onclick = accordionEnrollOnClick()' attribute to the element. Remember to set 'enroll_callback = FALSE' to turn off standard activation behavior (click action on header).

If you want the item to be disabled (like button) when the item is enrolled please also add 'class = activatorClass' to it.

Usage

```
accordionEnrollOnClick(prev = FALSE)

activatorClass
```

Arguments

`prev`	Should the current (FALSE) or previous (TRUE) item be enrolled? 'prev = TRUE' can be useful if the last accordion item is removed and we want to enroll the preceding item.

Format

An object of class `character` of length 1.

Value

'html' class string that can be used for defining i.e. 'onclick' attribute callback.

Character string - class name used for identifying accordion activator object.

Examples

```
if (interactive()) {
  library(shiny)
  activator <- function(disabled = FALSE) {
    tags$button(
      "Enroll", class = activatorClass, onclick = accordionEnrollOnClick(),
      disabled = if (isTRUE(disabled)) NA else NULL
    )
  }
  ui <- fluidPage(
    tags$head(tags$style(
```

```
      ".acc-header, .acc-content {border: 1px solid; border-radius: 5px;}"
    )),
    accordion(
      "acc",
      accordionItem(
        "first", div("Hello", activator(TRUE)), "There",
        enroll_callback = FALSE, active = TRUE
      ),
      accordionItem(
        "second", div("General", activator(FALSE)), "Kenobi",
        enroll_callback = FALSE
      )
    )
  )
  server <- function(input, output, session) {}

  shinyApp(ui, server)
}
```

---

accordionItem *Create accordion item*

Description

'accordionItem' allows to create new accordion item that can be passed directly to 'accordion' constructor or added on the fly with 'addAccordionItem'.

Usage

```
accordionItem(
  id,
  header,
  content,
  class = NULL,
  enroll_callback = TRUE,
  active = FALSE,
  header_class = NULL,
  content_class = NULL,
  ...
)

addAccordionItem(
  accordionId,
  accordionItem,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments

id	Unique id of accordion item.
header	Accordion item header.
content	Accordion item content.
class	Class passed to accordion container.
enroll_callback	It 'TRUE', click action on header will enroll the accordion item (and collapse the other existing ones). See accordionEnrollOnClick to see how configure custom on-click enroll element.
active	Should item be enrolled?
header_class	Additional class passed to header container.
content_class	Additional class passed to content container.
...	Extra elements passed to accordion container (before the first accordion item).
accordionId	Id of accordion component where item should be added.
accordionItem	Accordion item to be added.
session	Shiny Session object.

Value

Nested list of 'shiny.tag' objects, defining accordion item - its header and content, or no return value in case of using 'addAccordionItem' method.

Examples

```
if (interactive()) {
   library(shiny)
   ui <- fluidPage(
     actionButton("new", "New"),
     accordion(
       "acc",
       accordionItem("first", "Hello", "There", active = TRUE),
       accordionItem("second", "General", "Kenobi")
     )
   )
   server <- function(input, output, session) {}
   shinyApp(ui, server)

  # Accordion with custom styling of header and content (and dynamically added items).
  library(shiny)

  styled_item <- function(id, header_text, content_text, active = FALSE) {
    accordionItem(
      id, header_text, content_text, active = active,
      header_class = "acc-header", content_class = "acc-content"
    )
  }
  ui <- fluidPage(
```

```
    tags$head(tags$style(
      ".acc-header, .acc-content {border: 1px solid; border-radius: 5px;}"
    )),
    actionButton("new", "New"),
    accordion(
      "acc",
      styled_item("first", "Hello", "There", TRUE),
      styled_item("second", "General Kenobi", "There")
    )
  )
  server <- function(input, output, session) {
    observeEvent(input$new, {
      addAccordionItem(
        "acc",
        styled_item(
          sample(letters, 1), "I've Been Trained In Your Jedi Arts",
          "By Count Dooku", TRUE
        )
      )
    })
  }

  shinyApp(ui, server)
}
```

`animation` *Define an animation*

Description

Creates an ‘animation‘ object for usage in runAnimation.

Usage

```
animation(effect, delay = 0, duration = 1000)
```

Arguments

`effect`	Animation effect used name to be applied. Check .cssEffects object for possible options.
`delay`	Delay of animation start (in milliseconds).
`duration`	Duration of animation (in milliseconds).

Value

A named list with class ‘animation‘.

commonInput *Merge multiple input controllers into one*

Description

Select which input controllers should be treated as one. Use ‘commonInput‘ to group selected controllers or ‘commonInputs‘ to group multiple controllers at once.

Usage

```
commonInput(inputId, controller, block = TRUE, ignoreIds = NULL)

commonInputs(inputId, ..., block = TRUE, ignoreIds = NULL)
```

Arguments

inputId	Id to be used to send the grouped controllers input values to application server.
controller	Shiny input controller e.g. ‘shiny::sliderInput‘ or ‘shinyWidgets::pickerInput‘.
block	Should the ‘controller‘ input value be sent to the server independently?
ignoreIds	Precise input IDs of bindings that should be ignored. Leave NULL (default) to catch all.
...	Input controllers to be grouped in case of using ‘commonInputs‘.

Examples

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

  ui <- fluidPage(
    commonInput("val", selectInput("letter", "Letter", letters)),
    commonInput("val", numericInput("number", "Number", min = 0, max = 10, value = 1)),
    commonInputs(
      "val2",
      selectInput("letter2", "Letter", letters),
      numericInput("number2", "Number", min = 0, max = 10, value = 1)
    )
  )

  server <- function(input, output, session) {
    observeEvent(input$val, {
      print(input$val)
    })
    observeEvent(input$val2, {
      print(input$val2)
    })
  }

  shinyApp(ui, server)
}
```

custom-callbacks *Helpful methods for custom callback setup*

Description

Can be used as a ‘true‘ or ‘false‘ argument for custom method of js_calls.

Usage

```
runAnimation(..., ignoreInit = TRUE)
```

Arguments

...	Animation object(s) created with animation; if multiple animation objects are given then the animations will be chained.
ignoreInit	Should the animation be skipped when application is in initial state?

Examples

```
library(shiny)
library(shinyGizmo)
ui <- fluidPage(
    actionButton("value", "Click me", class = "btn-primary"),
    br(), br(),
    conditionalJS(
        tags$h1("Hello", style = "display: none;"),
        "input.value % 2 === 1",
        jsCalls$custom(
            true = runAnimation(animation("jello"), animation("swing")),
            false = runAnimation(animation("slideOutRight"))
        )
    )
)
server <- function(input, output, session) {}
if(interactive()) {
  shinyApp(ui, server)
}
```

jsCallOncePerFlush *Run JS when condition is met*

Description

'conditionalJS' is an extended version of conditionalPanel. The function allows to run selected or custom JS action when the provided condition is true or false.

To see the possible JS actions check jsCalls.

Optionally call 'jsCallOncePerFlush' in server to assure the call is run once per application flush cycle (see. https://github.com/rstudio/shiny/issues/3668). This prevents i.e. running animation multiple times when 'runAnimation(once = FALSE)' is used.

Usage

```
jsCallOncePerFlush(session)

conditionalJS(ui, condition, jsCall, once = TRUE, ns = shiny::NS(NULL))
```

Arguments

session	Shiny session object.
ui	A 'shiny.tag' element to which the JS callback should be attached.
condition	A JavaScript expression that will be evaluated repeatedly. When the evaluated 'condition' is true, 'jsCall''s true ('jsCall$true') callback is run, when false - 'jsCall$false' is executed in application browser.
jsCall	A list of two 'htmltools::JS' elements named 'true' and 'false' storing JS expressions. The 'true' object is evaluated when 'condition' is true, 'false' otherwise. In order to skip true/false callback assign it to NULL (or skip). Use 'this' object in the expressions to refer to the 'ui' object. See jsCalls for possible actions.
once	Should the JS action be called only when condition state changes?
ns	The NS object of the current module, if any.

Examples

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

  ui <- fluidPage(
    tags$style(".boldme {font-weight: bold;}"),
    sliderInput("value", "Value", min = 1, max = 10, value = 1),
    textOutput("slid_val"),
    conditionalJS(
      tags$button("Show me when slider value at least 3"),
      "input.value >= 3",
      jsCalls$show()
    ),
    hr(),
    conditionalJS(
      tags$button("Show me when value less than 3"),
      "input.value >= 3",
      jsCalls$show(when = FALSE)
    ),
```

```
  hr(),
  conditionalJS(
    tags$button("I'm disabled when value at least 4"),
    "input.value >= 4",
    jsCalls$disable()
  ),
  hr(),
  conditionalJS(
    tags$button("I'm disabled when value less than 4"),
    "input.value >= 4",
    jsCalls$disable(when = FALSE)
  ),
  hr(),
  conditionalJS(
    tags$button("I have class 'boldme' when value at least 5"),
    "input.value >= 5",
    jsCalls$attachClass("boldme")
  ),
  hr(),
  conditionalJS(
    tags$button("I change color when value at least 6"),
    "input.value >= 6",
    jsCalls$custom(
      true = "$(this).css('background-color', 'red');",
      false = "$(this).css('background-color', 'green');"
    )
  ),
  hr(),
  conditionalJS(
    tags$button("I change border when value at least 7"),
    "input.value >= 7",
    jsCalls$css(
      border = "dashed"
    )
  ),
  hr(),
  conditionalJS(
    tags$button("I'm disabled permanently when value at least 8"),
    "input.value >= 8",
    jsCalls$disable()["true"] # remove false condition
  ),
  hr(),
  conditionalJS(
    tags$button("I bounce when value at least 9"),
    "input.value >= 9",
    jsCalls$custom(true = runAnimation()),
    once = FALSE
  )
)

server <- function(input, output, session) {
  output$slid_val <- renderText({
    input$value
```

```
    })
    jsCallOncePerFlush(session)
  }

  shinyApp(ui, server)
}

if (interactive()) {
  library(shiny)
  library(shinyGizmo)

  ui <- fluidPage(
    textInput("name", "Name"),
    conditionalJS(
      actionButton("value", "Type name to enable the button"),
      "input.name != ''",
      jsCalls$disable(when = FALSE)
    )
  )

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

  shinyApp(ui, server)
}
```

`jsCalls` *List of JavaScript calls for 'conditionalJS'*

Description

Each 'jsCalls' function can be used as a 'jsCall' argument of conditionalJS. See js_calls for possible options.

You can apply multiple calls with using 'mergeCalls'.

Usage

```
jsCalls

mergeCalls(...)
```

Arguments

`...`	jsCalls to be merged.

Format

An object of class `list` of length 6.

Examples

```
conditionalJS(
  shiny::tags$button("Hello"),
  "input.value > 0",
  jsCalls$show()
)
if (interactive()) {
  library(shiny)

  ui <- fluidPage(
    tags$head(
      tags$script(
        "var update_attr = function(message) {",
        "$('#' + message.id).attr(message.attribute, message.value);",
        "}",
        "Shiny.addCustomMessageHandler('update_attr', update_attr);"
      )
    ),
    sidebarLayout(
      sidebarPanel(
        selectInput("effect", "Animation type", choices = .cssEffects)
      ),
      mainPanel(
        conditionalJS(
          ui = plotOutput("cars"),
          condition = "input.effect != ''",
          jsCall = jsCalls$custom(true = runAnimation(effect = "bounce")),
          once = FALSE
        )
      )
    )
  )

  server <- function(input, output, session) {
    output$cars <- renderPlot({
      plot(mtcars$mpg, mtcars$qsec)
    })
    observeEvent(input$effect, {
      session$sendCustomMessage(
        "update_attr",
       list(id = "cars", attribute = "data-call-if-true", value = runAnimation(input$effect))
      )
    })
  }


  shinyApp(ui, server)
}
```

`js_calls` *JavaScript calls for conditionalJS*

Description

The list of JavaScript calls that can be used as a 'jsCall' argument of conditionalJS. All the actions are reversible. E.g. when using 'disable' call and conditionalJS condition is false the opposite action to disable is called (removing disable attribute).

Usage

```
attachClass(class, when = TRUE)

disable(when = TRUE)

show(when = TRUE)

css(..., important = FALSE, when = TRUE)

animateVisibility(
  effectShow = "fadeIn",
  effectHide = "fadeOut",
  delay = 0,
  duration = 500,
  ignoreInit = TRUE,
  when = TRUE
)

custom(true = NULL, false = NULL)
```

Arguments

`class`	A css to be attached to (or detached from) the UI element.
`when`	Should the (primary) action be executed when 'condition' is TRUE (when = TRUE, default) or FALSE (when = FALSE).
`...`	Named style properties, where the name is the property name and the argument is the property value. See css for more details.
`important`	Should '!important' rule be attached to the added css?
`effectShow, effectHide`	Animation effects used for showing and hiding element. Check .cssEffects object for possible options.
`delay`	Delay of animation start (in milliseconds).
`duration`	Duration of animation (in milliseconds).
`ignoreInit`	Should the animation be skipped when application is in initial state?
`true, false`	JS callback that should be executed when condition is true or false. Can be custom JS (wrapped into JS) or one of the custom-callbacks.

Details

The currently offered actions:

- attachClass Add provided class to the UI element.
- disable Add disable attribute to the UI element - usually results with disabling the input controller.
- show Show/hide an element with a help of ‘visibility:hidden‘ rule. Comparing to conditionalPanel (which uses display:none) results with rendering an output even if hidden.
- css Add css (inline) rule to the UI object. When condition is false, the rule is removed.
- animateVisibility Show/hide an element in an animated way.
- custom Define custom true and false callback.

`modal-operations` *Show and hide modal from the application server*

Description

Show and hide modal from the application server

Usage

```
hideModalUI(modalId, session = shiny::getDefaultReactiveDomain())

showModalUI(modalId, session = shiny::getDefaultReactiveDomain())
```

Arguments

`modalId`	Id of the modal to show/hide.
`session`	Shiny session object.

Value

No return value, used for side effect.

`modalDialogUI` *Create modal in UI application part*

Description

Contrary to modalDialog the function allows to define modal in UI application structure. The modal can be opened with ‘modalButtonUI‘ placed anywhere in the application.

Usage

```
modalDialogUI(
  modalId,
  ...,
  button = modalButtonUI(modalId, "Open Modal"),
  title = NULL,
  footer = shiny::modalButton("Dismiss"),
  size = c("m", "s", "l", "xl"),
  easyClose = FALSE,
  fade = TRUE,
  backdrop = TRUE
)

modalButtonUI(modalId, label, icon = NULL, width = NULL, ...)
```

Arguments

`modalId`	Id of the modal.
`...`	Additional properties added to button.
`button`	Visible button placed in modal DOM structure, responsible for opening it. Set ‘NULL‘ to have no button included.
`title`	An optional title for the modal dialog.
`footer`	UI for modal dialog footer.
`size`	of the modal dialog. Can be "s", "m" (default), "l" or "xl".
`easyClose`	Set ‘TRUE‘ to enable closing modal with clicking outside it.
`fade`	Should fade-in animation be turned on?
`backdrop`	Set ‘FALSE‘ to turn on background covering area outside modal dialog.
`label`	Modal button label.
`icon`	Modal button icon.
`width`	Button width.

Value

Nested list of ‘shiny.tag‘ objects defining html structure of modal dialog, or single ‘shiny.tag‘ object in case of using ‘modalButtonUI‘ method.

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      modalDialogUI("mdl", "Hello")
    ),
    server = function(input, output, session) {}
  )

  library(shiny)
  shinyApp(
    ui = fluidPage(
      modalDialogUI("mdl", "Hello", button = NULL),
      hr(),
      modalButtonUI("mdl", "Open Modal From Here")
    ),
    server = function(input, output, session) {}
  )
}
```

`pickCheckboxInput` *Select set of active checkbox groups and their values*

Description

The component is connection of dropdown (pickerInput) (or virtualSelectInput) and set of checkbox groups (checkboxGroupInput).

When specific value is selected in dropdown, the related checkbox group becomes active and becomes visible to the user.

Usage

```
pickCheckboxInput(
  inputId,
  label,
  choices,
  choicesNames = pickCheckboxNames(choices),
  choicesLabels = pickCheckboxLabels(choices),
  selected = NULL,
  max_groups = length(choices),
  ...
)

vsCheckboxInput(
  inputId,
  label,
```

```
  choices,
  choicesNames = pickCheckboxNames(choices),
  choicesLabels = pickCheckboxLabels(choices),
  selected = NULL,
  max_groups = length(choices),
  ...
)

updatePickCheckboxInput(
  session,
  inputId,
  choices,
  choicesNames,
  choicesLabels,
  selected
)

updateVsCheckboxInput(
  session,
  inputId,
  choices,
  choicesNames,
  choicesLabels,
  selected
)
```

Arguments

inputId	Id of 'pickCheckboxInput' component.
label	The component label.
choices	Named list of values. Each element defines a separate checkbox group. The element name defines checkbox group id, whereas its value set of values that should be available in the related checkbox group.
choicesNames	Named list of values (with the same names as 'choices'). Each element value defines what labels should be displayed for each checkbox group. See pickCheckboxNamesAndLabels.
choicesLabels	Named vector storing labels for each checkbox group. The parameter is also used to display values in component dropdown. See pickCheckboxNamesAndLabels.
selected	The initial value or value to be updated. Subset of 'choices'.
max_groups	Number of maximum number of checkboxes allowed in the component. Used to limit amount of new checkbox groups added with 'updatePickCheckboxInput'.
...	Extra parameters passed to pickerInput or virtualSelectInput in case of usage pickCheckboxInput or vsCheckboxInput respectively.
session	Shiny session object.

Value

Nested list of ‘shiny.tag‘ objects, defining html structure of the input, or no value in case of usage of ‘updatePickCheckboxInput‘ method.

Examples

```
# Possible choices and selected configurations

# Choices as list of unnamed options
# Names are the same as values in the component (if not precised elsewhere)
choices_unnamed <- list(
  fruits = c("orange", "apple", "lemon"),
  vegetables = c("potato", "carrot", "broccoli")
)
# selected only fruits plus orange one within
selected_unnamed <- list(
  fruits = c("orange")
)
# Names for each group precised separately
choices_names = list(
  fruits = c("Orange", "Apple", "Lemon"),
  vegetables = c("Potato", "Carrot", "Broccoli")
)

# Choices as list of named options
# Names are treated as checkbox options labels
choices_named <- list(
  fruits = c("Orange" = "orange", "Apple" = "apple", "Lemon" = "lemon"),
  vegetables = c("Potato" = "potato", "Carrot" = "carrot", "Broccoli" = "broccoli")
)
# selected: fruits plus orange and vegetables carrot
selected_named <- list(
  fruits = c("orange"),
  vegetables= c("carrot")
)

# Same but vegetables selected but empty
# Set group as NA to no options checked (same effect in server input)
selected_named_empty <- list(
  fruits = c("orange"),
  vegetables = NA
)

# Specifying picker and group labels ("key" = "name" rule)
choices_labels <- list("fruits" = "Fruits", "vegetables" = "Vegetables")

if (interactive()) {
  library(shiny)

  ui <- fluidPage(
    sidebarLayout(sidebarPanel(
      pickCheckboxInput(
```

```
      "pci1", "1. No names at all",
      choices = choices_unnamed, selected = selected_unnamed
    ), hr(),
    pickCheckboxInput(
      "pci2", "2. Names provided as `choicesNames`",
    choices = choices_unnamed, selected = selected_unnamed, choicesNames = choices_names
    ), hr(),
    pickCheckboxInput(
      "pci3", "3. Names provided directly in choices",
      choices = choices_named, selected = selected_named
    ), hr(),
    pickCheckboxInput(
      "pci4", "4. Group as NA value to select group (without any choices)",
      choices = choices_named, selected = selected_named_empty
    ), hr(),
    pickCheckboxInput(
      "pci5", "5. Group names provided as `choicesLabels`",
     choices = choices_named, selected = selected_named_empty, choicesLabels = choices_labels
    )
   ),
   mainPanel(
     verbatimTextOutput("out1"),
     verbatimTextOutput("out2"),
     verbatimTextOutput("out3"),
     verbatimTextOutput("out4"),
     verbatimTextOutput("out5")
   ))
 )
 server <- function(input, output, session) {
   output$out1 <- renderPrint({print("Result 1."); input$pci1})
   output$out2 <- renderPrint({print("Result 2."); input$pci2})
   output$out3 <- renderPrint({print("Result 3."); input$pci3})
   output$out4 <- renderPrint({print("Result 4."); input$pci4})
   output$out5 <- renderPrint({print("Result 5."); input$pci5})
 }
 shinyApp(ui, server)
}
```

`pickCheckboxNamesAndLabels` *Generate names and labels*

Description

Two functions extracting group names and options labels from defined choices.

Usage

```
pickCheckboxNames(choices)
```

```
pickCheckboxLabels(choices)
```

Arguments

choices	linkpickCheckboxInput choices list.

Value

Named list object defining labels for component checkbox options, or named vector storing labels for each checkbox.

Examples

```
choices_unnamed <- list(
  fruits = c("orange", "apple", "lemon"),
  vegetables = c("potato", "carrot", "broccoli")
)
pickCheckboxNames(choices_unnamed)
pickCheckboxLabels(choices_unnamed)

choices_named <- list(
  fruits = c("Orange" = "orange", "Apple" = "apple", "Lemon" = "lemon"),
  vegetables = c("Potato" = "potato", "Carrot" = "carrot", "Broccoli" = "broccoli")
)
pickCheckboxNames(choices_named)
pickCheckboxLabels(choices_named)
```

---

shinyGizmo-imports *Objects imported from other packages*

Description

These objects are imported from other packages. Follow the links to their documentation: JS.

---

textArea *Text area component*

Description

Contrary to textAreaInput the component is not a binding itself (doesn't send input to the server). Thanks to that, the component can store much more text value without slowing down the application.

If you want to access the component value on request please use valueButton.

Usage

```
textArea(
  inputId,
  value,
  label,
  width = "100%",
  height = "200px",
  resize = "default",
  readonly = FALSE,
  ...
)

updateTextArea(session, inputId, value = NULL)
```

Arguments

inputId	Id of component. This is stored as ‘data-id‘ attribute to omit automatic binding of the element (into textAreaInput).
value	Initial text area value or value to be updated.
label	Text area label.
width	Width of input area.
height	Height of input area.
resize	Text are directions where input field can be resized. Possible options are "default", "both", "none", "vertical" and "horizontal".
readonly	If TRUE, providing custom values will be turned off.
...	Extra arguments passed to `textarea` tag form tags.
session	Shiny session object.

Value

Nested list of ‘shiny.tag‘ objects defining html structure of the component, or no value in case of usage of ‘updateTextArea‘ method.

---

valueButton *Take value from selected UI element*

Description

The components creates button or link that allows to take any value (or attribute) sourced from DOM element existing in Shiny application and pass it to application server.

Usage

```
valueButton(
  inputId,
  label,
  selector,
  attribute = "value",
  icon = NULL,
  width = NULL,
  try_binding = TRUE,
  ...
)

valueLink(
  inputId,
  label,
  selector,
  attribute = "value",
  icon = NULL,
  try_binding = TRUE,
  ...
)
```

Arguments

inputId	Id of the button. The value can be accessed in server with ‘input[[inputId]]‘.
label	Button label.
selector	CSS selector of element the attribute should be taken from. Set ‘"window"‘ or ‘"document"‘ to access application ‘window‘ or ‘document‘ object.
attribute	Name of the attribute that should be taken from desired element. For nested properties use ‘.‘, e.g. ‘style.width‘ or ‘navigator.appName‘.
icon	Icon included in button.
width	Width of the button.
try_binding	When ‘TRUE‘ and ‘selector‘ points to Shiny Binding and ‘attribute == "value"‘ it tries to convert sourced input value using registered ‘inputHandler‘.
...	Extra attributes passed to ‘button‘ or ‘a‘ tag.

Value

A ‘shiny.tag‘ class object defining html structure of the button.

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      tags$textarea(id = "txt"),
```

```
      valueButton("val", "Take textarea value", "#txt", attribute = "value")
    ),
    server = function(input, output, session) {
      observeEvent(input$val, print(input$val))
    }
  )
}
```

Index