Package 'shiny.semantic'

May 9, 2026

Type Package

Title Semantic UI Support for Shiny

Version 0.5.1

Description Creating a great user interface for your Shiny apps can be a hassle, especially if you want to work purely in R and don't want to use, for instance HTML templates. This package adds support for a powerful UI library Fomantic UI - <https://fomantic-ui.com/> (before Semantic). It also supports universal UI input binding that works with various DOM elements.

License MIT + file LICENSE

URL https://appsilon.github.io/shiny.semantic/, https://github.com/Appsilon/shiny.semantic

BugReports https://github.com/Appsilon/shiny.semantic/issues

Imports glue, grDevices, htmltools (>= 0.2.6), htmlwidgets (>= 0.8), jsonlite, magrittr, purrr (>= 0.2.2), R6, semantic.assets (>= 1.1.0), shiny (>= 0.12.1), stats

Suggests covr, chromote, dplyr, DT, gapminder, knitr, leaflet, lintr, markdown, mockery, plotly, rcmdcheck, rmarkdown, testthat, shinytest2, tibble, withr

VignetteBuilder knitr

Encoding UTF-8

Language en-US

RoxygenNote 7.3.1

NeedsCompilation no

Author Filip Stachura [aut],
Dominik Krzeminski [aut],
Krystian Igras [aut],
Adam Forys [aut],
Paweł Przytuła [aut],
Jakub Chojna [aut],
Olga Mierzwa-Sulima [aut],

Jakub Nowicki [aut, cre],
Tymoteusz Makowski [aut],
Ashley Baldry [ctb],
Pedro Manuel Coutinho da Silva [ctb],
Kamil Żyła [ctb],
Rabii Bouhestine [ctb],
Federico Rivandeira [ctb],
Appsilon Sp. z o.o. [cph]

Maintainer Jakub Nowicki <opensource+kuba@appsilon.com>

Repository CRAN

Date/Publication 2024-04-05 15:10:02 UTC

Contents

`accordion` *Accordion UI*

Description

In accordion you may display a list of elements that can be hidden or shown with one click.

Usage

```
accordion(
  accordion_list,
  fluid = TRUE,
  active_title = "",
  styled = TRUE,
  custom_style = ""
)
```

Arguments

`accordion_list`	list with lists with fields: ‘title‘ and ‘content‘
`fluid`	if accordion is fluid then it takes width of parent div
`active_title`	if active title matches ‘title‘ from `accordion_list` then this element is active by default
`styled`	if switched of then raw style (no boxes) is used
`custom_style`	character with custom style added to CSS of accordion (advanced use)

Value

shiny tag list with accordion UI

Examples

```
if (interactive()) {
library(shiny)
library(shiny.semantic)
accordion_content <- list(
  list(title = "AA", content = h2("a a a a")),
  list(title = "BB", content = p("b b b b"))
)
shinyApp(
  ui = semanticPage(
    accordion(accordion_content, fluid = F, active_title = "AA",
              custom_style = "background: #babade;")
  ),
  server = function(input, output) {}
)
}
```

`action_button` *Action button*

Description

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

Usage

```
action_button(input_id, label, icon = NULL, width = NULL, ...)

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

Arguments

`input_id`	The input slot that will be used to access the value.
`label`	The contents of the button - 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.
`...`	Named attributes to be applied to the button or remaining parameters passed to button, like `class`.
`inputId`	the same as `input_id`

Examples

```
if (interactive()){
library(shiny)
library(shiny.semantic)
ui <- shinyUI(semanticPage(
  actionButton("action_button", "Press Me!"),
  textOutput("button_output")
))
server <- function(input, output, session) {
  output$button_output <- renderText(as.character(input$action_button))
}
shinyApp(ui, server)
}
```

button *Create Semantic UI Button*

Description

Create Semantic UI Button

Usage

```
button(input_id, label, icon = NULL, class = NULL, ...)
```

Arguments

input_id	The input slot that will be used to access the value.
label	The contents of the button or link
icon	An optional icon() to appear on the button.
class	An optional attribute to be added to the button's class. If used paramters like color, size are ignored.
...	Named attributes to be applied to the button

Examples

```
if (interactive()){
library(shiny)
library(shiny.semantic)
ui <- semanticPage(
  shinyUI(
    button("simple_button", "Press Me!")
  )
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
}
```

calendar *Create Semantic UI Calendar*

Description

This creates a default calendar input using Semantic UI. The input is available under input[[input_id]].

This function updates the date on a calendar

Usage

```
calendar(
  input_id,
  value = NULL,
  placeholder = NULL,
  type = "date",
  min = NA,
  max = NA
)

update_calendar(session, input_id, value = NULL, min = NULL, max = NULL)
```

Arguments

input_id	ID of the calendar that will be updated
value	Initial value of the numeric input.
placeholder	Text visible in the input when nothing is inputted.
type	Select from 'year', 'month', 'date' and 'time'
min	Minimum allowed value.
max	Maximum allowed value.
session	The session object passed to function given to shinyServer.

Examples

```
# Basic calendar
if (interactive()) {

  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(
    semanticPage(
      title = "Calendar example",
      calendar("date"),
      p("Selected date:"),
      textOutput("selected_date")
    )
  )

   server <- shinyServer(function(input, output, session) {
     output$selected_date <- renderText(
       as.character(input$date)
     )
   })

   shinyApp(ui = ui, server = server)
 }

## Not run:
```

```
# Calendar with max and min
calendar(
  name = "date_finish",
  placeholder = "Select End Date",
  min = "2019-01-01",
  max = "2020-01-01"
)

# Selecting month
calendar(
  name = "month",
  type = "month"
)

## End(Not run)
```

`card` *Create Semantic UI card tag*

Description

This creates a card tag using Semantic UI styles.

Usage

```
card(..., class = "")
```

Arguments

`...`	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
`class`	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    card(
      div(class="content",
          div(class="header", "Elliot Fu"),
          div(class="meta", "Friend"),
          div(class="description", "Elliot Fu is a film-maker from New York.")
      )
    )
  ))
```

```
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

---

cards *Create Semantic UI cards tag*

Description

This creates a cards tag using Semantic UI styles.

Usage

```
cards(..., class = "")
```

Arguments

...	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
class	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    cards(
      class = "two",
      card(
        div(class="content",
            div(class="header", "Elliot Fu"),
            div(class="meta", "Friend"),
            div(class="description", "Elliot Fu is a film-maker from New York.")
        )
      ),
      card(
        div(class="content",
            div(class="header", "John Bean"),
            div(class="meta", "Friend"),
            div(class="description", "John Bean is a film-maker from London.")
        )
      )
    )
```

```
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

---

checkbox_input *Create Semantic UI checkbox*

Description

Create Semantic UI checkbox

Usage

```
checkbox_input(
  input_id,
  label = "",
  type = NULL,
  is_marked = TRUE,
  style = NULL
)

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

toggle(input_id, label = "", is_marked = TRUE, style = NULL)
```

Arguments

input_id	Input name. Reactive value is available under input[[name]].
label	Text to be displayed with checkbox.
type	Type of checkbox: NULL, 'toggle'
is_marked	Defines if checkbox should be marked. Default TRUE.
style	Style of the widget.
inputId	same as input_id
value	same as is_marked
width	The width of the input (currently not supported, but check style)

Details

The inputs are updateable by using updateCheckboxInput.

The following types are allowed:

- NULL The standard checkbox (default)
- toggle Each checkbox has a toggle form
- slider Each checkbox has a simple slider form

Examples

```
if (interactive()){
  ui <- shinyUI(
    semanticPage(
      p("Simple checkbox:"),
      checkbox_input("example", "Check me", is_marked = FALSE),
      p(),
      p("Simple toggle:"),
      toggle("tog1", "My Label", TRUE)
    )
  )
  server <- function(input, output, session) {
    observeEvent(input$tog1, {
      print(input$tog1)
    })
  }
  shinyApp(ui, server)
}
```

---

check_proper_color *Check if color is set from Fomantic-UI palette*

Description

Check if color is set from Fomantic-UI palette

Usage

```
check_proper_color(color)
```

Arguments

color	character with color name

Value

Error when color does not belong to palette

Examples

```
check_proper_color("blue")
```

COLOR_PALETTE *Semantic colors*

Description

https://github.com/Semantic-Org/Semantic-UI/blob/master/src/themes/default/globals/site.variables

Usage

```
COLOR_PALETTE
```

Format

An object of class `character` of length 13.

counter_button *Counter Button*

Description

Creates a counter button whose value increments by one each time it is pressed.

Usage

```
counter_button(
  input_id,
  label = "",
  icon = NULL,
  value = 0,
  color = "",
  size = "",
  big_mark = " "
)
```

Arguments

`input_id`	The `input` slot that will be used to access the value.
`label`	the content of the item to display
`icon`	an optional `icon()` to appear on the button.
`value`	initial rating value (integer)
`color`	character with semantic color
`size`	character with size of the button, eg. "medium", "big"
`big_mark`	big numbers separator

Value

counter button object

Examples

```
if (interactive()) {
library(shiny)
library(shiny.semantic)
ui <-semanticPage(
     counter_button("counter", "My Counter Button",
                    icon = icon("world"),
                    size = "big", color = "purple")
 )
server <- function(input, output) {
 observeEvent(input$counter,{
   print(input$counter)
  })
 }
shinyApp(ui, server)
}
```

`create_modal` *Allows for the creation of modals in the server side without being tied to a specific HTML element.*

Description

Allows for the creation of modals in the server side without being tied to a specific HTML element.

Usage

```
create_modal(
  ui_modal,
  show = TRUE,
  session = shiny::getDefaultReactiveDomain()
)

showModal(ui, session = shiny::getDefaultReactiveDomain())
```

Arguments

`ui_modal`	HTML containing the modal.
`show`	If the modal should only be created or open when called (open by default).
`session`	Current session.
`ui`	Same as `ui_modal` in show modal

See Also

modal

date_input *Define simple date input with Semantic UI styling*

Description

Define simple date input with Semantic UI styling

Usage

```
date_input(
  input_id,
  label = NULL,
  value = NULL,
  min = NULL,
  max = NULL,
  style = NULL,
  icon_name = "calendar"
)

dateInput(
  inputId,
  label = NULL,
  icon = NULL,
  value = NULL,
  min = NULL,
  max = NULL,
  width = NULL,
  ...
)
```

Arguments

input_id	Input id.
label	Label to be displayed with date input.
value	Default date chosen for input.
min	Minimum date that can be selected.
max	Maximum date that can be selected.
style	Css style for widget.
icon_name	Icon that should be displayed on widget.
inputId	Input id.
icon	Icon that should be displayed on widget.
width	character width of the object
...	other arguments

Examples

```
if (interactive()) {
# Below example shows how to implement simple date range input using \code{date_input}

library(shiny)
library(shiny.semantic)

ui <- shinyUI(
  semanticPage(
    title = "Date range example",
    uiOutput("date_range"),
    p("Selected dates:"),
    textOutput("selected_dates")
  )
)

server <- shinyServer(function(input, output, session) {
  output$date_range <- renderUI({
    tagList(
      tags$div(tags$div(HTML("From")),
               date_input("date_from", value = Sys.Date() - 30, style = "width: 10%;")),
      tags$div(tags$div(HTML("To")),
               date_input("date_to", value = Sys.Date(), style = "width: 10%;"))
    )
  })

  output$selected_dates <- renderPrint({
    c(input$date_from, input$date_to)
  })
})

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

`display_grid` *Display grid template in a browser for easy debugging*

Description

Display grid template in a browser for easy debugging

Usage

```
display_grid(grid_template)
```

Arguments

`grid_template`	generated by grid_template() function

Details

Opens a browser and displays grid template with styled border to highlight existing areas.

Warning: CSS can't be displayed in RStudio viewer pane. CSS grid is supported only by modern browsers. You can see list of supported browsers here: https://www.w3schools.com/css/css_grid.asp

---

`dropdown_input` *Create dropdown Semantic UI component*

Description

This creates a default *dropdown_input* using Semantic UI styles with Shiny input. Dropdown is already initialized and available under input[[input_id]].

Usage

```
dropdown_input(
  input_id,
  choices,
  choices_value = choices,
  default_text = "Select",
  value = NULL,
  type = "selection fluid"
)
```

Arguments

`input_id`	Input name. Reactive value is available under input[[input_id]].
`choices`	All available options one can select from.
`choices_value`	What reactive value should be used for corresponding choice.
`default_text`	Text to be visible on dropdown when nothing is selected.
`value`	Pass value if you want to initialize selection for dropdown.
`type`	Change depending what type of dropdown is wanted.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(shiny.semantic)
ui <- semanticPage(
  title = "Dropdown example",
  dropdown_input("simple_dropdown", LETTERS, value = "A"),
  p("Selected letter:"),
  textOutput("dropdown")
)
server <- function(input, output) {
```

```
  output$dropdown <- renderText(input[["simple_dropdown"]])
}

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

---

dropdown_menu *Create Semantic UI Dropdown*

Description

This creates a dropdown using Semantic UI.

Usage

```
dropdown_menu(
  ...,
  class = "",
  name,
  is_menu_item = FALSE,
  dropdown_specs = list()
)
```

Arguments

...	Dropdown content.
class	class of the dropdown. Look at https://semantic-ui.com/modules/dropdown.html for all possibilities.
name	Unique name of the created dropdown.
is_menu_item	TRUE if the dropdown is a menu item. Default is FALSE.
dropdown_specs	A list of dropdown functionalities. Look at https://semantic-ui.com/modules/dropdown.html#/settings for all possibilities.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    dropdown_menu(
      "Dropdown menu",
      icon(class = "dropdown"),
      menu(
        menu_header("Header"),
        menu_divider(),
```

```
        menu_item("Option 1"),
        menu_item("Option 2")
      ),
      name = "dropdown_menu",
      dropdown_specs = list("duration: 500")
    )

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

  shinyApp(ui, server)
}
```

`field` *Create Semantic UI field tag*

Description

This creates a field tag using Semantic UI styles.

Usage

```
field(..., class = "")
```

Arguments

`...`	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
`class`	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    form(
      field(
        tags$label("Name"),
        text_input("name", value = "", type = "text", placeholder = "Enter Name...")
      ),
      # error field
      field(
        class = "error",
        tags$label("Name"),
```

```
      text_input("name", value = "", type = "text", placeholder = "Enter Name...")
    ),
    # disabled
    field(
      class = "disabled",
      tags$label("Name"),
      text_input("name", value = "", type = "text", placeholder = "Enter Name...")
    )
  )
))
server <- shinyServer(function(input, output) {
})

shinyApp(ui, server)
}
```

`fields` *Create Semantic UI fields tag*

Description

This creates a fields tag using Semantic UI styles.

Usage

```
fields(..., class = "")
```

Arguments

`...`	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
`class`	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    form(
      fields(class = "two",
             field(
               tags$label("Name"),
             text_input("name", value = "", type = "text", placeholder = "Enter Name...")
             ),
             field(
```

```
              tags$label("Surname"),
            text_input("surname", value = "", type = "text", placeholder = "Enter Surname...")
               ))
    )
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

`file_input` *Create Semantic UI File Input*

Description

This creates a default file input using Semantic UI. The input is available under `input[[input_id]]`.

Usage

```
file_input(
  input_id,
  label,
  multiple = FALSE,
  accept = NULL,
  button_label = "Browse...",
  type = NULL,
  placeholder = "no file selected",
  ...
)

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

Arguments

`input_id`, `inputId`	Input name. Reactive value is available under `input[[input_id]]`.
`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.
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.
button_label, buttonLabel	Display label for the button.
type	Input type specifying class attached to input container. See [Fomantic UI](https://fomantic-ui.com/collections/form.html) for details.
placeholder	Inner input label displayed when no file has been uploaded.
...	Other parameters passed from fileInput to file_input like type.
width	The width of the input, e.g. '400px', or '100%'.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)
  ui <- semanticPage(
    form(
      div(
        class = "ui grid",
        div(
          class = "four wide column",
          file_input("ex", "Select file"),
          header("File type selected:", textOutput("ex_file"))
        )
      )
    )
  )
  server <- function(input, output, session) {
    output$ex_file <- renderText({
      if (is.null(input)) return("No file uploaded")
      tools::file_ext(input$ex$datapath)
    })
  }
  shinyApp(ui, server)
}
```

---

flow_layout *Flow layout*

Description

Lays out elements in a left-to-right, top-to-bottom arrangement. The elements on a given row will be top-aligned with each other.

Usage

```
flow_layout(
  ...,
  cell_args = list(),
  min_cell_width = "208px",
  max_cell_width = "1fr",
  column_gap = "12px",
  row_gap = "0px"
)

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

Arguments

`...`	Unnamed arguments will become child elements of the layout. Named arguments will become HTML attributes on the outermost tag.
`cell_args`	Any additional attributes that should be used for each cell of the layout.
`min_cell_width`	The minimum width of the cells.
`max_cell_width`	The maximum width of the cells.
`column_gap`	The spacing between columns.
`row_gap`	The spacing between rows.
`cellArgs`	Same as `cell_args`.

Details

The width of the elements and spacing between them is configurable. Lengths can be given as numeric values (interpreted as pixels) or character values (interpreted as CSS lengths). With the default settings this layout closely resembles the `flowLayout` from Shiny.

Examples

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

form *Create Semantic UI form tag*

Description

This creates a form tag using Semantic UI styles.

Usage

```
form(..., class = "")
```

Arguments

...	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
class	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    form(
      field(
        tags$label("Text"),
        text_input("text_ex", value = "", type = "text", placeholder = "Enter Text...")
      )
    ),
    # loading form
    form(class = "loading form",
         field(
           tags$label("Text"),
          text_input("text_ex", value = "", type = "text", placeholder = "Enter Text...")
         )),
    # size variations mini form
    form(class = "mini",
         field(
           tags$label("Text"),
          text_input("text_ex", value = "", type = "text", placeholder = "Enter Text...")
         )),
    # massive
    form(class = "massive",
         field(
           tags$label("Text"),
          text_input("text_ex", value = "", type = "text", placeholder = "Enter Text...")
         ))
```

```
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

`grid` *Use CSS grid template in Shiny UI*

Description

Use CSS grid template in Shiny UI

Usage

```
grid(
  grid_template,
  id = paste(sample(letters, 5), collapse = ""),
  container_style = "",
  area_styles = list(),
  display_mode = FALSE,
  ...
)
```

Arguments

`grid_template`	grid template created with grid_template() function
`id`	id of grid
`container_style`	character - string of custom CSS for the main grid container
`area_styles`	list of custom CSS styles for provided area names
`display_mode`	replaces areas HTML content with <area name> text. Used by display_grid() function
`...`	areas HTML content provided by named arguments

Details

Grids can be nested.

Value

Rendered HTML ready to use by Shiny UI. See `htmltools::htmlTemplate()` for more details.

Examples

```
myGrid <- grid_template(default = list(
  areas = rbind(
    c("header", "header", "header"),
    c("menu",   "main",   "right1"),
    c("menu",   "main",   "right2")
  ),
  rows_height = c("50px", "auto", "100px"),
  cols_width = c("100px", "2fr", "1fr")
))

subGrid <- grid_template(default = list(
  areas = rbind(
    c("top_left", "top_right"),
    c("bottom_left", "bottom_right")
  ),
  rows_height = c("50%", "50%"),
  cols_width = c("50%", "50%")
))

if (interactive()){
library(shiny)
library(shiny.semantic)
shinyApp(
  ui = semanticPage(
    grid(myGrid,
         container_style = "border: 1px solid #f00",
         area_styles = list(header = "background: #0099f9",
                            menu = "border-right: 1px solid #0099f9"),
         header = div(shiny::tags$h1("Hello CSS Grid!")),
         menu = checkbox_input("example", "Check me", is_marked = FALSE),
         main = grid(subGrid,
                     top_left = calendar("my_calendar"),
                     top_right = div("hello 1"),
                     bottom_left = div("hello 2"),
                     bottom_right = div("hello 3")
         ),
         right1 = div(
           toggle("toggle", "let's toggle"),
           multiple_checkbox("mycheckbox", "mycheckbox",
                             c("option A","option B","option C"))),
         right2 = div("right 2")
    )
  ),
  server = shinyServer(function(input, output) {})
)
}
```

`grid_template` *Define a template of a CSS grid*

Description

Define a template of a CSS grid

Usage

```
grid_template(default = NULL, mobile = NULL)
```

Arguments

default	(required) Template for desktop: list(areas = [data.frame of character], rows_height = [vector of character], cols_width = [vector of character])
mobile	(optional) Template for mobile (screen width below 768px): list(areas = [data.frame of character], rows_height = [vector of character], cols_width = [vector of character])

Value

list(template = [character], area_names = [vector of character])

template - contains template that can be parsed by htmlTemplate() function

area_names - contains all unique area names used in grid definition

Examples

```
myGrid <- grid_template(
  default = list(
    areas = rbind(
      c("header", "header", "header"),
      c("menu",   "main",   "right1"),
      c("menu",   "main",   "right2")
    ),
    rows_height = c("50px", "auto", "100px"),
    cols_width = c("100px", "2fr", "1fr")
  ),
  mobile = list(
    areas = rbind(
      "header",
      "menu",
      "main",
      "right1",
      "right2"
    ),
    rows_height = c("50px", "50px", "auto", "150px", "150px"),
    cols_width = c("100%")
  )
)
if (interactive()) display_grid(myGrid)
subGrid <- grid_template(default = list(
  areas = rbind(
    c("top_left", "top_right"),
    c("bottom_left", "bottom_right")
```

```
  ),
  rows_height = c("50%", "50%"),
  cols_width = c("50%", "50%")
))

if (interactive()) display_grid(subGrid)
```

---

header *Create Semantic UI header*

Description

This creates a header with optional icon using Semantic UI styles.

Usage

```
header(title, description, icon = NULL)
```

Arguments

title	Header title
description	Subheader text
icon	Optional icon name

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    header(title = "Header with description", description = "Description"),
    header(title = "Header with icon", description = "Description", icon = "dog")
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

`horizontal_menu` *Horizontal menu*

Description

Renders UI with horizontal menu

Usage

```
horizontal_menu(menu_items, active_location = "", logo = NULL)
```

Arguments

`menu_items`	list with list that can have fields: "name" (mandatory), "link" and "icon"
`active_location`	active location of the menu (should match one from "link")
`logo`	optional argument that displays logo on the left of horizontal menu, can be character with image location, or shiny image object

Value

shiny div with horizontal menu

Examples

```
library(shiny.semantic)
menu_content <- list(
 list(name = "AA", link = "http://example.com", icon = "dog"),
 list(name = "BB", link = "#", icon="cat"),
 list(name = "CC")
)
if (interactive()){
  ui <- semanticPage(
   horizontal_menu(menu_content)
  )
  server <- function(input, output, session) {}
  shinyApp(ui, server)
}
```

icon *Create Semantic UI icon tag*

Description

This creates an icon tag using Semantic UI styles.

Usage

```
icon(class = "", ...)
```

Arguments

class	A name of an icon. Look at http://semantic-ui.com/elements/icon.html for all possibilities.
...	Other arguments to be added as attributes of the tag (e.g. style, class etc.)

Examples

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

ui <- function() {
  shinyUI(
    semanticPage(
      # Basic icon
      icon("home"),
      br(),
      # Different size
      icon("small home"),
      icon("large home"),
      br(),
      # Disabled icon
      icon("disabled home"),
      br(),
      # Loading icon
      icon("spinner loading"),
      br(),
      # Icon formatted as link
      icon("close link"),
      br(),
      # Flipped
      icon("horizontally flipped cloud"),
      icon("vertically flipped cloud"),
      br(),
      # Rotated
      icon("clockwise rotated cloud"),
      icon("counterclockwise rotated cloud"),
```

```
      br(),
      # Circular
      icon("circular home"),
      br(),
      # Bordered
      icon("bordered home"),
      br(),
      # Colored
      icon("red home"),
      br(),
      # inverted
      segment(class = "inverted", icon("inverted home"))
    )
  )
}

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

})

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

`label` *Create Semantic UI label tag*

Description

This creates a `div` or a tag with with class `ui label` using Semantic UI.

Usage

```
label(..., class = "", is_link = TRUE)
```

Arguments

`...`	Other arguments to be added such as content of the tag (text, icons) and/or attributes (style)
`class`	class of the label. Look at https://semantic-ui.com/elements/label.html for all possibilities.
`is_link`	If TRUE creates label with 'a' tag, otherwise with 'div' tag. #'

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)
```

```
  ui <- shinyUI(
    semanticPage(
      ## label
      label(icon = icon("mail icon"), 23),
      p(),
      ## pointing label
      field(
        text_input("ex", label = "", type = "text", placeholder = "Your name")),
      label("Please enter a valid name", class = "pointing red basic"),
      p(),
      ## tag
      label(class = "tag", "New"),
      label(class = "red tag", "Upcoming"),
      label(class =" teal tag","Featured"),
      ## ribbon
      segment(class = "ui raised segment",
              label(class = "ui red ribbon", "Overview"),
              "Text"),
      ## attached
      segment(class = "ui raised segment",
              label(class = "top attached", "HTML"),
              p("Text"))
    ))
  server <- function(input, output, session) {
  }
  shinyApp(ui, server)
}
```

`list_container` *Create Semantic UI list with header, description and icons*

Description

This creates a list with icons using Semantic UI

Usage

```
list_container(content_list, is_divided = FALSE)
```

Arguments

`content_list`	list of lists with fields: 'header' and/or 'description', 'icon' containing the list items headers, descriptions (one of these is mandatory) and icons. Icon column should contain strings with icon names available here: https://fomantic-ui.com/elements/icon.html
`is_divided`	If TRUE created list elements are divided

Examples

```
library(shiny)
library(shiny.semantic)
list_content <- list(
  list(header = "Head", description = "Lorem ipsum", icon = "cat"),
  list(header = "Head 2", icon = "tree"),
  list(description = "Lorem ipsum 2", icon = "dog")
)
if (interactive()){
  ui <- semanticPage(
    list_container(list_content, is_divided = TRUE)
 )
  server <- function(input, output) {}
  shinyApp(ui, server)
}
```

`menu` *Create Semantic UI Menu*

Description

This creates a menu using Semantic UI.

Usage

```
menu(..., class = "")
```

Arguments

`...`	Menu items to be created. Use menu_item function to create new menu item. Use dropdown_menu(is_menu_item = TRUE, ...) function to create new dropdown menu item. Use menu_header and menu_divider functions to customize menu format.
`class`	Class extension.Look at https://semantic-ui.com/collections/menu.html for all possibilities.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)

  ui <- function() {
    shinyUI(
      semanticPage(
        title = "My page",
        menu(menu_item("Menu"),
```

```
                dropdown_menu(
                  "Action",
                  menu(
                    menu_header(icon("file"), "File", is_item = FALSE),
                    menu_item(icon("wrench"), "Open"),
                    menu_item(icon("upload"), "Upload"),
                    menu_item(icon("remove"), "Upload"),
                    menu_divider(),
                    menu_header(icon("user"), "User", is_item = FALSE),
                    menu_item(icon("add user"), "Add"),
                    menu_item(icon("remove user"), "Remove")),
                  class = "",
                  name = "unique_name",
                  is_menu_item = TRUE),
                menu_item(icon("user"), "Profile", href = "#index", item_feature = "active"),
                 menu_item("Projects", href = "#projects"),
                 menu_item(icon("users"), "Team"),
                 menu(menu_item(icon("add icon"), "New tab"), class = "right"))
        )
      )
    }
    server <- shinyServer(function(input, output) {})
    shinyApp(ui = ui(), server = server)
  }
```

`menu_divider` *Create Semantic UI Divider Item*

Description

This creates a menu divider item using Semantic UI.

Usage

```
menu_divider(...)
```

Arguments

`...`	Other attributes of the divider such as style.

See Also

menu

menu_header *Create Semantic UI Header Item*

Description

This creates a dropdown header item using Semantic UI.

Usage

```
menu_header(..., is_item = TRUE)
```

Arguments

...	Content of the header: text, icons, etc.
is_item	If TRUE created header is item of Semantic UI Menu.

See Also

menu

menu_item *Create Semantic UI Menu Item*

Description

This creates a menu item using Semantic UI

Usage

```
menu_item(..., item_feature = "", style = NULL, href = NULL)
```

Arguments

...	Content of the menu item: text, icons or labels to be displayed.
item_feature	If required, add additional item feature like 'active', 'header', etc.
style	Style of the item, e.g. "text-align: center".
href	If NULL (default) menu_item is created with 'div' tag. Otherwise it is created with 'a' tag, and parameeter defines its href attribute.

See Also

menu

message_box *Create Semantic UI Message box*

Description

Create Semantic UI Message box

Usage

```
message_box(header, content, class = "", icon_name, closable = FALSE)
```

Arguments

header	Header of the message box
content	Content of the message box . If it is a vector, creates a list of vector's elements
class	class of the message. Look at https://semantic-ui.com/collections/message.html for all possibilities.
icon_name	If the message is of the type 'icon', specify the icon. Look at http://semantic-ui.com/elements/icon.html for all possibilities.
closable	Determines whether the message should be closable. Default is FALSE - not closable

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    message_box(header = "Main header", content = "text"),
    # message with icon
   message_box(class = "icon", header = "Main header", content = "text", icon_name = "dog"),
    # closable message
    message_box(header = "Main header", content = "text", closable =  TRUE),
    # floating
    message_box(class = "floating", header = "Main header", content = "text"),
    # compact
    message_box(class = "compact", header = "Main header", content = "text"),
    # warning
    message_box(class = "warning", header = "Warning", content = "text"),
    # info
    message_box(class = "info", header = "Info", content = "text")
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
```

```
}
```

`modal` *Create Semantic UI modal*

Description

This creates a modal using Semantic UI styles.

Usage

```
modal(
  ...,
  id = "",
  class = "",
  header = NULL,
  content = NULL,
  footer = div(class = "ui button positive", "OK"),
  target = NULL,
  settings = NULL,
  modal_tags = NULL
)

modalDialog(..., title = NULL, footer = NULL)
```

Arguments

`...`	Content elements to be added to the modal body. To change attributes of the container please check the ‘content‘ argument.
`id`	ID to be added to the modal div. Default "".
`class`	Classes except "ui modal" to be added to the modal. Semantic UI classes can be used. Default "".
`header`	Content to be displayed in the modal header. If given in form of a list, HTML attributes for the container can also be changed. Default "".
`content`	Content to be displayed in the modal body. If given in form of a list, HTML attributes for the container can also be changed. Default NULL.
`footer`	Content to be displayed in the modal footer. Usually for buttons. If given in form of a list, HTML attributes for the container can also be changed. Set NULL, to make empty.
`target`	Javascript selector for the element that will open the modal. Default NULL.
`settings`	list of vectors of Semantic UI settings to be added to the modal. Default NULL.
`modal_tags`	character with title for `modalDialog` - equivalent to header
`title`	title displayed in header in `modalDialog`

Examples

```
## Create a simple server modal
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- function() {
  shinyUI(
    semanticPage(
      actionButton("show", "Show modal dialog")
    )
  )
}

server = function(input, output) {
  observeEvent(input$show, {
    create_modal(modal(
      id = "simple-modal",
      header = h2("Important message"),
      "This is an important message!"
    ))
  })
}
shinyApp(ui, server)
}
## Create a simple UI modal

if (interactive()) {
library(shiny)
library(shiny.semantic)
ui <- function() {
  shinyUI(
    semanticPage(
      title = "Modal example - Static UI modal",
      div(id = "modal-open-button", class = "ui button", "Open Modal"),
      modal(
        div("Example content"),
        id = "example-modal",
        target = "modal-open-button"
      )
    )
  )
}

## Observe server side actions
library(shiny)
library(shiny.semantic)
ui <- function() {
  shinyUI(
    semanticPage(
      title = "Modal example - Server side actions",
      uiOutput("modalAction"),
```

```
      actionButton("show", "Show by calling show_modal")
    )
  )
}

server <- shinyServer(function(input, output) {
  observeEvent(input$show, {
    show_modal('action-example-modal')
  })
  observeEvent(input$hide, {
    hide_modal('action-example-modal')
  })

  output$modalAction <- renderUI({
    modal(
      actionButton("hide", "Hide by calling hide_modal"),
      id = "action-example-modal",
      header = "Modal example",
      footer = "",
      class = "tiny"
    )
  })
})
shinyApp(ui, server)
}

## Changing attributes of header and content.
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- function() {
  shinyUI(
    semanticPage(
      actionButton("show", "Show modal dialog")
    )
  )
}

server = function(input, output) {
  observeEvent(input$show, {
    create_modal(modal(
      id = "simple-modal",
      title = "Important message",
      header = list("!!!", style = "background: lightcoral"),
      content = list(style = "background: lightblue",
                     `data-custom` = "value", "This is an important message!"),
      p("This is also part of the content!")
    ))
  })
}
shinyApp(ui, server)
}
```

```
## Modal that closes automatically after specific time
if (interactive()) {
library(shiny)
library(shiny.semantic)
ui <- function() {
  shinyUI(
    semanticPage(
      actionButton("show", "Show modal dialog")
    )
  )
}

server <- shinyServer(function(input, output, session) {
  observeEvent(input$show, {
    create_modal(
      modal(
        id = "simple-modal",
        title = "Important message",
        header = "Example modal",
        content = "This modal will close after 3 sec.",
        footer = NULL,
      )
    )
    Sys.sleep(3)
    hide_modal(id = "simple-modal")
  })
})

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

`multiple_checkbox` *Create Semantic UI multiple checkbox*

Description

This creates a multiple checkbox using Semantic UI styles.

Usage

```
multiple_checkbox(
  input_id,
  label,
  choices,
  choices_value = choices,
  selected = NULL,
  position = "grouped",
  type = NULL,
```

```
  ...
)

multiple_radio(
  input_id,
  label,
  choices,
  choices_value = choices,
  selected = choices_value[1],
  position = "grouped",
  type = "radio",
  ...
)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
label	Text to be displayed with checkbox.
choices	Vector of labels to show checkboxes for.
choices_value	Vector of values that should be used for corresponding choice. If not specified, choices is used by default.
selected	The value(s) that should be chosen initially. If NULL the first one from choices is chosen.
position	Specified checkmarks setup. Can be grouped or inline.
type	Type of checkbox or radio.
...	Other arguments to be added as attributes of the tag (e.g. style, childrens etc.)

Details

The following types are allowed:

- NULL The standard checkbox (default)
- toggle Each checkbox has a toggle form
- slider Each checkbox has a simple slider form

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  # Checkbox
  library(shiny)
  library(shiny.semantic)

  ui <- function() {
      shinyUI(
        semanticPage(
          title = "Checkbox example",
          h1("Checkboxes"),
```

```
          multiple_checkbox("checkboxes", "Select Letters", LETTERS[1:6], selected = "A"),
           p("Selected letters:"),
           textOutput("selected_letters"),
           tags$br(),
           h1("Radioboxes"),
           multiple_radio("radioboxes", "Select Letter", LETTERS[1:6], selected = "A"),
           p("Selected letter:"),
           textOutput("selected_letter")
         )
       )
  }

  server <- shinyServer(function(input, output) {
     output$selected_letters <- renderText(paste(input$checkboxes, collapse = ", "))
     output$selected_letter <- renderText(input$radioboxes)
  })

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

---

numeric_input *Create Semantic UI Numeric Input*

Description

This creates a default numeric input using Semantic UI. The input is available under `input[[input_id]]`.

Usage

```
numeric_input(
  input_id,
  label,
  value = NULL,
  min = NA,
  max = NA,
  step = NA,
  type = NULL,
  icon = NULL,
  placeholder = NULL,
  ...
)

numericInput(
  inputId,
  label,
  value = NULL,
  min = NA,
  max = NA,
```

```
  step = NA,
  width = NULL,
  placeholder = NULL,
  ...
)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
label	Display label for the control, or NULL for no label.
value	Initial value of the numeric input.
min	Minimum allowed value.
max	Maximum allowed value.
step	Interval to use when stepping between min and max.
type	Input type specifying class attached to input container. See [Fomantic UI](https://fomantic-ui.com/collections/form.html) for details.
icon	Icon or label attached to numeric input.
placeholder	Inner input label displayed when no value is specified
...	Other parameters passed to numeric_input like type or icon.
inputId	The input slot that will be used to access the value.
width	The width of the input.

Details

Either ‘value‘ or ‘placeholder‘ should be defined. The inputs are updateable by using updateNumericInput.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)
  ui <- semanticPage(
    numeric_input("ex", "Select number", 10),
  )
  server <- function(input, output, session) {}
  shinyApp(ui, server)
}
```

`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")‘][shinyOptions] 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:

```
Progress$new(session = getDefaultReactiveDomain(), min = 0, max = 1, ...)
```

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‘.

`...` Arguments that may have been used for ‘shiny::Progress‘

Method `set()`**:** Updates the progress panel. When called the first time, the progress panel is displayed.

Usage:

```
Progress$set(value = NULL, message = 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).

`...` Arguments that may have been used for ‘shiny::Progress‘

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, ...)
```

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).

`...` Arguments that may have been used for ‘shiny::Progress‘

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

[with_progress()]

Examples

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

ui <- semanticPage(
  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')

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

shinyApp(ui, server)
}
```

---

progress *Create progress Semantic UI component*

Description

This creates a default progress using Semantic UI styles with Shiny input. Progress is already initialized and available under input[[input_id]].

Usage

```
progress(
  input_id,
  value = NULL,
  total = NULL,
  percent = NULL,
  progress_lab = FALSE,
  label = NULL,
  label_complete = NULL,
```

```
  size = "",
  class = NULL
)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
value	The initial value to be selected for the progress bar.
total	The maximum value that will be applied to the progress bar.
percent	The initial percentage to be selected for the progress bar.
progress_lab	Logical, would you like the percentage visible in the progress bar?
label	The label to be visible underneath the progress bar.
label_complete	The label to be visible underneath the progress bar when the bar is at 100%.
size	character with legal semantic size, eg. "medium", "huge", "tiny"
class	UI class of the progress bar.

Details

To initialize the progress bar, you can either choose `value` and `total`, or `percent`.

Examples

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

  library(shiny)
  library(shiny.semantic)
  ui <- function() {
      shinyUI(
        semanticPage(
          title = "Progress example",
          progress("progress", percent = 24, label = "{percent}% complete"),
          p("Progress completion:"),
          textOutput("progress")
       )
     )
  }
  server <- shinyServer(function(input, output) {
     output$progress <- renderText(input$progress)
  })

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

rating_input *Rating Input.*

Description

Crates rating component

Usage

```
rating_input(
  input_id,
  label = "",
  value = 0,
  max = 3,
  icon = "star",
  color = "yellow",
  size = ""
)
```

Arguments

input_id	The input slot that will be used to access the value.
label	the contents of the item to display
value	initial rating value
max	maximum value
icon	character with name of the icon or icon() that is an element of the rating
color	character with colour name
size	character with legal semantic size, eg. "medium", "huge", "tiny"

Value

rating object

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(
    semanticPage(
      rating_input("rate", "How do you like it?", max = 5,
                   icon = "heart", color = "yellow"),
    )
  )
```

```
  server <- function(input, output) {
    observeEvent(input$rate,{print(input$rate)})
  }
  shinyApp(ui = ui, server = server)
}
```

`register_search` *Register search api url*

Description

Calls Shiny session's registerDataObj to create REST API. Publishes any R object as a URL endpoint that is unique to Shiny session. search_query 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). search_query will be called with these values whenever an HTTP request is made to the URL endpoint. The return value of search_query should be a list of list or a dataframe. Note that different semantic components expect specific JSON fields to be present in order to work correctly. Check components documentation for details.

Usage

```
register_search(session, data, search_query)
```

Arguments

`session`	Shiny server session
`data`	Data (the value that is passed into registerDataObj)
`search_query`	Function providing a response as a list of lists or dataframe of search results.

Examples

```
if (interactive()) {
  library(shiny)
  library(tibble)
  library(shiny.semantic)
  shinyApp(
    ui = semanticPage(
      textInput("txt", "Enter the car name (or subset of name)"),
      textOutput("api_url"),
      uiOutput("open_url")
    ),
    server = function(input, output, session) {
      api_response <- function(data, q) {
        has_matching <- function(field) {
          grepl(toupper(q), toupper(field), fixed = TRUE)
        }
        dplyr::filter(data, has_matching(car))
      }
```

```
        search_api_url <- register_search(session,
                             tibble::rownames_to_column(mtcars, "car"), api_response)

        output$api_url <- renderText({
         glue::glue(
          "Registered API url: ",
          "{session$clientData$url_protocol}//{session$clientData$url_hostname}",
          ":{session$clientData$url_port}/",
          "{search_api_url}&q={input$txt}"
          )
        })

        output$open_url <- renderUI({
          tags$a(
            "Open", class = "ui button",
            href = glue::glue("./{search_api_url}&q={input$txt}"), target = "_blank"
          )
        })
      }
    )
  }
```

---

render_menu_link *Render menu link*

Description

This function renders horizontal menu item.

Usage

```
render_menu_link(location, title, active_location = "", icon = NULL)
```

Arguments

location	character url with location
title	name of the page
active_location	name of the active subpage (if matches location then it gets highlighted), default empty ("")
icon	non-mandatory parameter with icon name

Value

shiny tag link

See Also

horizontal_menu

Examples

```
render_menu_link("#subpage1", "SUBPAGE")
```

---

search_field *Create search field Semantic UI component*

Description

This creates a default search field using Semantic UI styles with Shiny input. Search field is already initialized and available under input[[input_id]]. Search will automatically route to the named API endpoint provided as parameter. API response is expected to be a JSON with property fields ‘title‘ and ‘description‘. See https://semantic-ui.com/modules/search.html#behaviors for more details.

Usage

```
search_field(input_id, search_api_url, default_text = "Search", value = "")
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
search_api_url	Register custom API url with server JSON Response containing fields ‘title‘ and ‘description‘.
default_text	Text to be visible on serach field when nothing is selected.
value	Pass value if you want to initialize selection for search field.

Examples

```
## Only run examples in interactive R sessions
## Not run:
if (interactive()) {
library(shiny)
library(shiny.semantic)
library(gapminder)
library(dplyr)

ui <- function() {
  shinyUI(
    semanticPage(
      title = "Dropdown example",
      p("Search country:"),
      uiOutput("search_country"),
      p("Selected country:"),
      textOutput("selected_country")
   )
```

```
  )
}

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

 search_api <- function(gapminder, q) {
   has_matching <- function(field) {
     startsWith(field, q)
   }
   gapminder %>%
     mutate(country = as.character(country)) %>%
     select(country) %>%
     unique %>%
     filter(has_matching(country)) %>%
     head(5) %>%
     transmute(title = country,
               description = country)
 }

 search_api_url <- register_search(session, gapminder, search_api)
 output$search_letters <- shiny::renderUI(
   search_field("search_result", search_api_url)
 )
 output$selected_country <- renderText(input[["search_result"]])
})
}

shinyApp(ui = ui(), server = server)

## End(Not run)
```

`search_selection_api` *Add Semantic UI search selection dropdown based on REST API*

Description

Define the (multiple) search selection dropdown input for retrieving remote selection menu content from an API endpoint. API response is expected to be a JSON with property fields ‘name‘ and ‘value‘. Using a search selection dropdown allows to search more easily through large lists.

Usage

```
search_selection_api(
  input_id,
  search_api_url,
  multiple = FALSE,
  default_text = "Select"
)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
search_api_url	Register API url with server JSON Response containing fields ‘name‘ and ‘value‘.
multiple	TRUE if the dropdown should allow multiple selections, FALSE otherwise (default FALSE).
default_text	Text to be visible on dropdown when nothing is selected.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
   library(shiny)
   library(shiny.semantic)
   library(gapminder)
   library(dplyr)

   ui <- function() {
    shinyUI(
      semanticPage(
        title = "Dropdown example",
        uiOutput("search_letters"),
        p("Selected letter:"),
        textOutput("selected_letters")
      )
    )
  }

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

   search_api <- function(gapminder, q) {
     has_matching <- function(field) {
        startsWith(field, q)
      }
      gapminder %>%
        mutate(country = as.character(country)) %>%
        select(country) %>%
        unique %>%
        filter(has_matching(country)) %>%
        head(5) %>%
          transmute(name = country,
                  value = country)
    }

    search_api_url <- shiny.semantic::register_search(session,
                                                      gapminder,
                                                      search_api)
    output$search_letters <- shiny::renderUI(
      search_selection_api("search_result", search_api_url, multiple = TRUE)
    )
    output$selected_letters <- renderText(input[["search_result"]])
  })
```

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

---

search_selection_choices *Add Semantic UI search selection dropdown based on provided choices*

Description

Define the (multiple) search selection dropdown input component serving search options using provided choices.

Usage

```
search_selection_choices(
  input_id,
  choices,
  value = NULL,
  multiple = FALSE,
  default_text = "Select",
  groups = NULL,
  dropdown_settings = list(forceSelection = FALSE)
)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
choices	Vector or a list of choices to search through.
value	String with default values to set when initialize the component. Values should be delimited with a comma when multiple to set. Default NULL.
multiple	TRUE if the dropdown should allow multiple selections, FALSE otherwise (default FALSE).
default_text	Text to be visible on dropdown when nothing is selected.
groups	Vector of length equal to choices, specifying to which group the choice belongs. Specifying the parameter enables group dropdown search implementation.
dropdown_settings	Settings passed to dropdown() semantic-ui method. See https://semantic-ui.com/modules/dropdown.html#

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)

  ui <- function() {
    shinyUI(
      semanticPage(
        title = "Dropdown example",
        uiOutput("search_letters"),
        p("Selected letter:"),
        textOutput("selected_letters")
      )
    )
  }

  server <- shinyServer(function(input, output, session) {
    choices <- LETTERS
    output$search_letters <- shiny::renderUI(
      search_selection_choices("search_result", choices, multiple = TRUE)
    )
    output$selected_letters <- renderText(input[["search_result"]])
  })

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

---

segment *Create Semantic UI segment*

Description

This creates a segment using Semantic UI styles.

Usage

```
segment(..., class = "")
```

Arguments

...	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
class	Additional classes to add to html tag.

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(semanticPage(
    segment(),
    # placeholder
    segment(class = "placeholder segment"),
    # raised
    segment(class = "raised segment"),
    # stacked
    segment(class = "stacked segment"),
    #  piled
    segment(class = "piled segment")
  ))
  server <- shinyServer(function(input, output) {
  })

  shinyApp(ui, server)
}
```

`selectInput` *Create a select list input control*

Description

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

Usage

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

Arguments

`inputId`	The input slot that will be used to access the value.
`label`	Display label for the control, or NULL for no label.

choices	List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user.
selected	The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
multiple	Is selection of multiple items allowed?
width	The width of the input.
...	Arguments passed to dropdown_input.

Examples

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

  library(shiny.semantic)

  # basic example
  shinyApp(
    ui = semanticPage(
      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)
    }
  )
}
```

`semanticPage` *Semantic UI page*

Description

This creates a Semantic page for use in a Shiny app.

Usage

```
semanticPage(
  ...,
  title = "",
  theme = NULL,
  suppress_bootstrap = TRUE,
  margin = "10px"
)
```

Arguments

...	Other arguments to be added as attributes of the main div tag wrapper (e.g. style, class etc.)
title	A title to display in the browser's title bar.
theme	Theme name or path. Full list of supported themes you will find in semantic.assets::SUPPORTED_THEME or at http://semantic-ui-forest.com/themes.
suppress_bootstrap	boolean flag that supresses bootstrap when turned on
margin	character with body margin size

Details

Inside, it uses two crucial options:

(1) shiny.minified with a logical value, tells whether it should attach min or full semnatic css or js (TRUE by default). (2) shiny.custom.semantic if this option has not NULL character semanticPage takes dependencies from custom css and js files specified in this path (NULL by default). Depending on shiny.minified value the folder should contain either "min" or standard version. The folder should contain: semantic.css and semantic.js files, or semantic.min.css and semantic.min.js in shiny.minified = TRUE mode.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- semanticPage(
  title = "Hello Shiny Semantic!",
  tags$label("Number of observations:"),
  slider_input("obs", value = 500, min = 0, max = 1000),
  segment(
    plotOutput("dist_plot")
  )
)

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

shinyApp(ui, server)
}
```

`semantic_DT` *Create Semantic DT Table*

Description

This creates DT table styled with Semantic UI.

Usage

```
semantic_DT(
  ...,
  options = list(),
  style = "semanticui",
  class = "ui small compact table"
)
```

Arguments

`...`	datatable parameters, check `?DT::datatable` to learn more.
`options`	datatable options, check `?DT::datatable` to learn more.
`style`	datatable style, check `?DT::datatable` to learn more.
`class`	datatable class, check `?DT::datatable` to learn more.

Examples

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

 ui <- semanticPage(
   semantic_DTOutput("table")
 )
 server <- function(input, output, session) {
   output$table <- DT::renderDataTable(
     semantic_DT(iris)
   )
 }
 shinyApp(ui, server)
}
```

semantic_DTOutput *Semantic DT Output*

Description

Semantic DT Output

Usage

```
semantic_DTOutput(...)
```

Arguments

...	datatable parameters, check ?DT::datatable to learn more.

Value

DT Output with semanitc style

shiny_input *Create universal Shiny input binding*

Description

Universal binding for Shiny input on custom user interface. Using this function one can create various inputs ranging from text, numerical, date, dropdowns, etc. Value of this input is extracted via jQuery using $().val() function and default exposed as serialized JSON to the Shiny server. If you want to change type of exposed input value specify it via type param. Currently list of supported types is "JSON" (default) and "text".

Usage

```
shiny_input(input_id, shiny_ui, value = NULL, type = "JSON")
```

Arguments

input_id	String with name of this input. Access to this input within server code is normal with input[[input_id]].
shiny_ui	UI of HTML component presenting this input to the users. This UI should allow to extract its value with jQuery $().val() function.
value	An optional argument with value that should be set for this input. Can be used to store persisten input valus in dynamic UIs.
type	Type of input value (could be "JSON" or "text").

Examples

```
library(shiny)
library(shiny.semantic)
# Create a week field
uirender(
  tagList(
    div(class = "ui icon input",
        style = NULL,
        "",
        shiny_input(
          "my_id",
          tags$input(type = "week", name = "my_id", min = NULL, max = NULL),
          value = NULL,
          type = "text"),
        icon("calendar"))
  )
)
```

`shiny_text_input` *Create universal Shiny text input binding*

Description

Universal binding for Shiny text input on custom user interface. Value of this input is extracted via jQuery using $().val() function. This function is just a simple binding over shiny_input. Please take a look at shiny_input documentation for more information.

Usage

```
shiny_text_input(...)
```

Arguments

`...`	Possible arguments are the same as in shiny_input() method: input_id, shiny_ui, value. Type is already predefined as "text"

Examples

```
library(shiny)
library(shiny.semantic)
# Create a color picker
uirender(
  tagList(
    div(class = "ui input",
        style = NULL,
        "Color picker",
        shiny_text_input(
          "my_id",
```

```
        tags$input(type = "color", name = "my_id", value = "#ff0000"))
    )
  ))
```

---

show_modal *Show, Hide or Remove Semantic UI modal*

Description

This displays a hidden Semantic UI modal.

Usage

```
show_modal(id, session = shiny::getDefaultReactiveDomain(), asis = TRUE)

remove_modal(id, session = shiny::getDefaultReactiveDomain(), asis = TRUE)

remove_all_modals(session = shiny::getDefaultReactiveDomain())

removeModal(session = shiny::getDefaultReactiveDomain())

hide_modal(id, session = shiny::getDefaultReactiveDomain(), asis = TRUE)
```

Arguments

id	ID of the modal that will be displayed.
session	The session object passed to function given to shinyServer.
asis	A boolean indicating if the id must be handled as is (TRUE) or FALSE if it meants to be namespaced

See Also

modal

---

sidebar_panel *Creates div containing children elements of sidebar panel*

Description

Creates div containing children elements of sidebar panel

Creates div containing children elements of main panel

Creates grid layout composed of sidebar and main panels

Usage

```
sidebar_panel(..., width = 1)

main_panel(..., width = 3)

sidebar_layout(
  sidebar_panel,
  main_panel,
  mirrored = FALSE,
  min_height = "auto",
  container_style = "",
  area_styles = list(sidebar_panel = "", main_panel = "")
)

sidebarPanel(..., width = 6)

mainPanel(..., width = 10)

sidebarLayout(
  sidebarPanel,
  mainPanel,
  position = c("left", "right"),
  fluid = TRUE
)
```

Arguments

`...`	Container's children elements
`width`	Width of main panel container as relative value
`sidebar_panel`	Sidebar panel component
`main_panel`	Main panel component
`mirrored`	If TRUE sidebar is located on the right side, if FALSE - on the left side (default)
`min_height`	Sidebar layout container keeps the minimum height, if specified. It should be formatted as a string with css units
`container_style`	CSS declarations for grid container
`area_styles`	List of CSS declarations for each grid area inside
`sidebarPanel`	same as `sidebar_panel`
`mainPanel`	same as `main_panel`
`position`	vector with position of sidebar elements in order sidebar, main
`fluid`	TRUE to use fluid layout; FALSE to use fixed layout.

Value

Container with sidebar and main panels

Examples

```
if (interactive()){
  library(shiny)
  library(shiny.semantic)
  ui <- semanticPage(
    titlePanel("Hello Shiny!"),
    sidebar_layout(
      sidebar_panel(
        shiny.semantic::sliderInput("obs",
                                    "Number of observations:",
                                    min = 0,
                                    max = 1000,
                                    value = 500),
                                    width = 3
      ),
      main_panel(
        plotOutput("distPlot"),
        width = 4
      ),
      mirrored = TRUE
    )
  )
  server <- function(input, output) {
    output$distPlot <- renderPlot({
      hist(rnorm(input$obs))
    })
  }
  shinyApp(ui, server)
}
```

---

single_step *Creates a single step to be used inside of a list of steps by the steps function*

Description

Creates a single step to be used inside of a list of steps by the steps function

Usage

```
single_step(
  id,
  title,
  description = NULL,
  icon_class = NULL,
  step_class = NULL
)
```

Arguments

id	The input slot that will be used to access the value.
title	A character that will be the title of the ste
description	A character that will fill the description of the step
icon_class	A character which will be correpond to a fomantic icon class to be used in the step
step_class	A character representing a class to be passed to the step

See Also

steps

SIZE_LEVELS *Allowed sizes*

Description

Allowed sizes

Usage

```
SIZE_LEVELS
```

Format

An object of class `character` of length 7.

---

slider_input *Create Semantic UI Slider / Range*

Description

This creates a slider input using Semantic UI. Slider is already initialized and available under `input[[input_id]]`. Use Range for range of values.

Usage

```
slider_input(
  input_id,
  value,
  min,
  max,
  step = 1,
  class = "labeled",
  custom_ticks = NULL
)

sliderInput(
  inputId,
  label,
  min,
  max,
  value,
  step = 1,
  width = NULL,
  ticks = TRUE,
  ...
)

range_input(input_id, value, value2, min, max, step = 1, class = NULL)
```

Arguments

input_id	Input name. Reactive value is available under input[[input_id]].
value	The initial value to be selected for the sldier (lower value if using range).
min	The minimum value allowed to be selected for the slider.
max	The maximum value allowed to be selected for the slider.
step	The interval between each selectable value of the slider.
class	UI class of the slider. Can include "labeled" and "ticked".
custom_ticks	A vector of custom labels to be added to the slider. Will ignore min and max
inputId	Input name.
label	Display label for the control, or NULL for no label.
width	character with width of slider.
ticks	FALSE to hide tick marks, TRUE to show them according to some simple heuristics
...	additional arguments
value2	The initial upper value of the slider.

Details

Use update_slider to update the slider/range within the shiny session.

See Also

update_slider for input updates, https://fomantic-ui.com/modules/slider.html for preset classes.

Examples

```
if (interactive()) {
  # Slider example
  library(shiny)
  library(shiny.semantic)

  ui <- shinyUI(
    semanticPage(
      title = "Slider example",
      tags$br(),
      slider_input("slider", 10, 0, 20, class = "labeled ticked"),
      p("Selected value:"),
      textOutput("slider")
    )
  )
  server <- shinyServer(function(input, output, session) {
    output$slider <- renderText(input$slider)
  })
  shinyApp(ui = ui, server = server)

  # Custom ticks slider
  ui <- shinyUI(
    semanticPage(
      title = "Slider example",
      tags$br(),
     slider_input("slider_ticks", "F", custom_ticks = LETTERS, class = "labeled ticked"),
      p("Selected value:"),
      textOutput("slider_ticks")
    )
  )
  server <- shinyServer(function(input, output, session) {
    output$slider_ticks <- renderText(input$slider_ticks)
  })
  shinyApp(ui = ui, server = server)

  # Range example
  ui <- shinyUI(
    semanticPage(
      title = "Range example",
      tags$br(),
      range_input("range", 10, 15, 0, 20),
      p("Selected values:"),
      textOutput("range")
    )
  )
  server <- shinyServer(function(input, output, session) {
    output$range <- renderText(paste(input$range, collapse = " - "))
```

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

`split_layout` *Split layout*

Description

Lays out elements horizontally, dividing the available horizontal space into equal parts (by default) or specified by parameters.

Usage

```
split_layout(..., cell_widths = NULL, cell_args = "", style = NULL)

splitLayout(..., cellWidths = NULL, cellArgs = "", style = NULL)
```

Arguments

`...`	Unnamed arguments will become child elements of the layout.
`cell_widths`	Character or numeric vector indicating the widths of the individual cells. Recycling will be used if needed.
`cell_args`	character with additional attributes that should be used for each cell of the layout.
`style`	character with style of outer box surrounding all elements
`cellWidths`	same as `cell_widths`
`cellArgs`	same as `cell_args`

Value

split layout grid object

Examples

```
if (interactive()) {
  #' 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 <- semanticPage(
    split_layout(
      plotOutput("plot1"),
```

```
    plotOutput("plot2")
  )
)
shinyApp(ui, server)
#' Custom widths
ui <- semanticPage(
  split_layout(cell_widths = 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 <- semanticPage(
  split_layout(
  cell_widths = 300,
  cell_args = "padding: 6px;",
  style = "border: 1px solid silver;",
  plotOutput("plot1"),
  plotOutput("plot2"),
  plotOutput("plot3")
)
)
shinyApp(ui, server)
}
```

`steps` *Show steps*

Description

Show steps

Usage

```
steps(id, steps_list, class = NULL)
```

Arguments

`id`	ID of the Steps that will be displayed.
`steps_list`	A list of steps generated by single_steps.
`class`	(Optional) A character string with the semantic class to be added to the steps element.

See Also

single_steps

Examples

```
if (interactive()) {
 library(shiny)
 library(shiny.semantic)
 ui <- semanticPage(
 title = "Steps Example",
 shiny::tagList(
   h2("Steps example"),
   shiny.semantic::steps(
     id = "steps",
     steps_list = list(
         single_step(
           id = "step_1",
           title = "Step 1",
           description = "It's night?",
           icon_class = "moon"
         ),
         single_step(
           id = "step_2",
           title = "Step 2",
           description = "Order some food",
           icon_class = "bug"
         ),
         single_step(id = "step_3",
                     title = "Step 3",
                     description = "Feed the Kiwi",
                     icon_class = "kiwi bird"
                    )
     )
   ),
   h3("Actions"),
   shiny.semantic::action_button("step_1_complete", "Make it night"),
   shiny.semantic::action_button("step_2_complete", "Call the insects"),
   shiny.semantic::action_button("step_3_complete", "Feed the Kiwi"),
   shiny.semantic::action_button("hungry_kiwi", "Kiwi is hungry again"),
 )
)

 server <- function(input, output, session) {
   observeEvent(input$step_1_complete, {
     toggle_step_state("step_1")
   })

   observeEvent(input$step_2_complete, {
     toggle_step_state("step_2")
   })

   observeEvent(input$step_3_complete, {
     toggle_step_state("step_3")
   })

   observeEvent(input$hungry_kiwi, {
```

```
    toggle_step_state("step_1", FALSE)
    toggle_step_state("step_2", FALSE)
    toggle_step_state("step_3", FALSE)
  })

}

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

---

tabset *Create Semantic UI tabs*

Description

This creates tabs with content using Semantic UI styles.

Usage

```
tabset(
  tabs,
  active = NULL,
  id = generate_random_id("menu"),
  menu_class = "top attached tabular",
  tab_content_class = "bottom attached grid segment"
)
```

Arguments

tabs	A list of tabs. Each tab is a list of three elements - first element defines menu item, second element defines tab content, third optional element defines tab id.
active	Id of the active tab. If NULL first tab will be active.
id	Id of the menu element (default: randomly generated id)
menu_class	Class for the menu element (default: "top attached tabular")
tab_content_class	Class for the tab content (default: "bottom attached segment")

Details

You may access active tab id with `input$<id>`.

See Also

update_tabset

Examples

```
## Only run examples in interactive R sessions
if (interactive()){
  library(shiny)
  library(shiny.semantic)

  ui <- semanticPage(
    tabset(tabs =
             list(
               list(menu = "First Tab", content = "Tab 1"),
               list(menu = "Second Tab", content = "Tab 2", id = "second_tab")
             ),
           active = "second_tab",
           id = "exampletabset"
    ),
    h2("Active Tab:"),
    textOutput("activetab")
  )
  server <- function(input, output) {
      output$activetab <- renderText(input$exampletabset)
  }
  shinyApp(ui, server)
}
```

`textAreaInput` *Create a semantic Text Area input*

Description

Create a text area input control for entry of unstructured text values.

Usage

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

Arguments

`inputId`	Input name. Reactive value is available under `input[[input_id]]`.
`label`	character with label put above the input
`value`	Pass value if you want to have default text.
`width`	The width of the input, eg. "40px"
`placeholder`	Text visible in the input when nothing is inputted.

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
ui <- semanticPage(
  textAreaInput("a", "Area:", value = "200", width = "200px"),
  verbatimTextOutput("value")
)
server <- function(input, output, session) {
  output$value <- renderText({ input$a })
}
shinyApp(ui, server)
}
```

---

text_input *Create Semantic UI Text Input*

Description

This creates a default text input using Semantic UI. The input is available under `input[[input_id]]`.

Usage

```
text_input(
  input_id,
  label = NULL,
  value = "",
  type = "text",
  placeholder = NULL,
  attribs = list()
)

textInput(
  inputId,
  label,
  value = "",
  width = NULL,
  placeholder = NULL,
  type = "text"
)
```

Arguments

`input_id`	Input name. Reactive value is available under `input[[input_id]]`.
`label`	character with label put on the left from the input
`value`	Pass value if you want to have default text.
`type`	Change depending what type of input is wanted. See details for options.

placeholder	Text visible in the input when nothing is inputted.
attribs	A named list of attributes to assign to the input.
inputId	Input name. The same as input_id.
width	The width of the input, eg. "40px"

Details

The following type s are allowed:

- text The standard input
- textarea An extended space for text
- password A censored version of the text input
- email A special version of the text input specific for email addresses
- url A special version of the text input specific for URLs
- tel A special version of the text input specific for telephone numbers

The inputs are updateable by using updateTextInput or updateTextAreaInput if type = "textarea".

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
  library(shiny)
  library(shiny.semantic)
  ui <- semanticPage(
    uiinput(
      text_input("ex", label = "Your text", type = "text", placeholder = "Enter Text")
    )
  )
  server <- function(input, output, session) {
 }
 shinyApp(ui, server)
 }
```

---

theme_selector *Themes changer dropdown*

Description

Themes changer dropdown

Usage

```
theme_selector(input_id = "theme", label = "Choose theme")
```

Arguments

input_id	Id of dropdown. input[[input_id]] returns the currently selected theme.
label	Dropdown label.

Examples

```
if (interactive()) {
library(shiny)
 library(shiny.semantic)
  ui <- semanticPage(
    theme = "superhero",
    actionButton("action_button", "Press Me!"),
    textOutput("button_output"),
    theme_selector(),
    textOutput("theme")
  )
  server <- function(input, output, session) {
    output$button_output <- renderText(as.character(input$action_button))
    output$theme <- renderText(as.character(input$theme))
  }
  shinyApp(ui, server)
}
```

---

toast *Show and remove Semantic UI toast*

Description

These functions either create or remove a toast notifications with Semantic UI styling.

Usage

```
toast(
  message,
  title = NULL,
  action = NULL,
  duration = 3,
  id = NULL,
  class = "",
  toast_tags = NULL,
  session = shiny::getDefaultReactiveDomain()
)

close_toast(id, session = shiny::getDefaultReactiveDomain())

showNotification(
  ui,
```

```
  action = NULL,
  duration = 5,
  closeButton = TRUE,
  id = NULL,
  type = c("default", "message", "warning", "error"),
  session = getDefaultReactiveDomain(),
  ...
)

removeNotification(id, session = shiny::getDefaultReactiveDomain())
```

Arguments

message	Content of the message.
title	A title given to the toast. Defauly is empty ("").
action	A list of lists containing settings for buttons/options to select within the
duration	Length in seconds for the toast to appear, default is 3 seconds. To make it not automatically close, set to 0.
id	A unique identifier for the notification. It is optional for `toast`, but required for `close_toast`.
class	Classes except "ui toast" to be added to the toast. Semantic UI classes can be used. Default "".
toast_tags	Other toast elements. Default NULL.
session	Session object to send notification to.
ui	Content of the toast.
closeButton	Logical, should a close icon appear on the toast?
type	Type of toast
...	Arguments that can be passed to `toast`

See Also

https://fomantic-ui.com/modules/toast

Examples

```
## Create a simple server toast
library(shiny)
library(shiny.semantic)

ui <- function() {
  shinyUI(
    semanticPage(
      actionButton("show", "Show toast")
    )
  )
}
```

```
server = function(input, output) {
  observeEvent(input$show, {
    toast(
      "This is an important message!"
    )
  })
}
if (interactive()) shinyApp(ui, server)

## Create a toast with options
ui <- semanticPage(
  actionButton("show", "Show"),
)
server <- function(input, output) {
  observeEvent(input$show, {
    toast(
      title  = "Question",
      "Do you want to see more?",
      duration = 0,
      action = list(
        list(
          text = "OK", class = "green", icon = "check",
          click = ("(function() { $('body').toast({message:'Yes clicked'}); })")
        ),
        list(
          text = "No", class = "red", icon = "times",
          click = ("(function() { $('body').toast({message:'No ticked'}); })")
        )
      )
    )
  })
}

if (interactive()) shinyApp(ui, server)

## Closing a toast
ui <- semanticPage(
  action_button("show", "Show"),
  action_button("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 <- toast(paste("Message", n), duration = NULL)
    ids <<- c(ids, id)
    n <<- n + 1
  })
```

```
  observeEvent(input$remove, {
    if (length(ids) > 0)
      close_toast(ids[1])
    ids <<- ids[-1]
  })
}

if (interactive()) shinyApp(ui, server)
```

`toggle_step_state` *Toggle step state*

Description

Toggle step state

Usage

```
toggle_step_state(id, state = TRUE, automatic_steps = TRUE, asis = TRUE)
```

Arguments

`id`	ID of step to be toggled
`state`	State of the step, `TRUE` stands for enabled
`automatic_steps`	Whether to toggle focus of next step automatically
`asis`	When used inside of Shiny module, `TRUE` will disable adding the namespace to `id`

See Also

steps

`uiinput` *Create Semantic UI Input*

Description

This creates an input shell for the actual input

Usage

```
uiinput(..., class = "")
```

Arguments

...	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
class	Additional classes to add to html tag.

See Also

text_input

Examples

```
#' ## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- semanticPage(
  uiinput(icon("dog"),
          numeric_input("input", value = 0, label = "")
  )
)

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

shinyApp(ui, server)
}
```

---

uirender *Render semanticui htmlwidget*

Description

htmlwidget that adds semanticui dependencies and renders in viewer or rmarkdown.

Usage

```
uirender(ui, width = NULL, height = NULL, element_id = NULL)
```

Arguments

ui	UI, which will be wrapped in an htmlwidget.
width	Fixed width for widget (in css units). The default is NULL, which results in intelligent automatic sizing.
height	Fixed height for widget (in css units). The default is NULL, which results in intelligent automatic sizing.
element_id	Use an explicit element ID for the widget (rather than an automatically generated one).

Examples

```
library(shiny)
library(shiny.semantic)
uirender(
  card(
    div(
      class="content",
      div(class="header", "Elliot Fu"),
      div(class="meta", "Friend"),
      div(class="description", "Elliot Fu is a film-maker from New York.")
    )
  )
)
```

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

Description

Update selecInput widget

Usage

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

Arguments

`session`	The session object passed to function given to shinyServer.
`inputId`	The id of the input object.
`label`	The label to set for the input object.
`choices`	List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user.
`selected`	The initially selected value (or multiple values if multiple = TRUE). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.

Examples

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

  ui <- semanticPage(
    p("The checkbox group controls the select input"),
    multiple_checkbox("checkboxes", "Input checkbox",
                      c("Item A", "Item B", "Item C")),
    selectInput("inSelect", "Select input",
                c("Item A", "Item B"))
  )

  server <- function(input, output, session) {
    observe({
      x <- input$checkboxes

      # 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(input$checkboxes, collapse = ", "),
                        choices = x,
                        selected = tail(x, 1)
      )
    })
  }

  shinyApp(ui, server)
}
```

---

update_action_button *Change the label or icon of an action button on the client*

Description

Change the label or icon of an action button on the client

Usage

```
update_action_button(session, input_id, label = NULL, icon = NULL)

updateActionButton(session, inputId, label = NULL, icon = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
input_id	The id of the input object.
label	The label to set for the input object.
icon	The icon to set for the input object. To remove the current icon, use icon=character(0)
inputId	the same as input_id

Examples

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

ui <- semanticPage(
  actionButton("update", "Update button"),
  br(),
  actionButton("go_button", "Go")
)

server <- function(input, output, session) {
  observe({
    req(input$update)

    # Updates go_button's label and icon
    updateActionButton(session, "go_button",
                       label = "New label",
                       icon = icon("calendar"))

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

---

update_dropdown_input *Update dropdown Semantic UI component*

Description

Change the value of a dropdown_input input on the client.

Usage

```
update_dropdown_input(
  session,
  input_id,
  choices = NULL,
```

```
  choices_value = choices,
  value = NULL
)
```

Arguments

session	The session object passed to function given to shinyServer.
input_id	The id of the input object
choices	All available options one can select from. If no need to update then leave as NULL
choices_value	What reactive value should be used for corresponding choice.
value	A value to update dropdown to. Defaults to NULL. • a value from choices updates the selection • character(0) and "" clear the selection • NULL: – clears the selection if choices is provided – otherwise, NULL does not change the selection • a value not found in choices does not change the selection

Examples

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

  ui <- semanticPage(
    title = "Dropdown example",
   dropdown_input("simple_dropdown", LETTERS[1:5], value = "A", type = "selection multiple"),
    p("Selected letter:"),
    textOutput("selected_letter"),
    shiny.semantic::actionButton("simple_button", "Update input to D")
  )

  server <- function(input, output, session) {
   output$selected_letter <- renderText(paste(input[["simple_dropdown"]], collapse = ", "))

    observeEvent(input$simple_button, {
      update_dropdown_input(session, "simple_dropdown", value = "D")
    })
  }

  shinyApp(ui, server)
}
```

update_multiple_checkbox *Update checkbox Semantic UI component*

Description

Change the value of a `multiple_checkbox` input on the client.

Usage

```
update_multiple_checkbox(
  session = getDefaultReactiveDomain(),
  input_id,
  choices = NULL,
  choices_value = choices,
  selected = NULL,
  label = NULL
)

update_multiple_radio(
  session = getDefaultReactiveDomain(),
  input_id,
  choices = NULL,
  choices_value = choices,
  selected = NULL,
  label = NULL
)
```

Arguments

session	The session object passed to function given to shinyServer.
input_id	The id of the input object
choices	All available options one can select from. If no need to update then leave as NULL
choices_value	What reactive value should be used for corresponding choice.
selected	The initially selected value.
label	The label linked to the input

Examples

```
if (interactive()) {

library(shiny)
library(shiny.semantic)

ui <- function() {
```

```
  shinyUI(
    semanticPage(
      title = "Checkbox example",
      form(
        multiple_checkbox(
       "simple_checkbox", "Letters:", LETTERS[1:5], selected = c("A", "C"), type = "slider"
        )
      ),
      p("Selected letter:"),
      textOutput("selected_letter"),
      shiny.semantic::actionButton("simple_button", "Update input to D")
    )
  )
}

server <- shinyServer(function(input, output, session) {
 output$selected_letter <- renderText(paste(input[["simple_checkbox"]], collapse = ", "))

  observeEvent(input$simple_button, {
    update_multiple_checkbox(session, "simple_checkbox", selected = "D")
  })
})

shinyApp(ui = ui(), server = server)

}
```

`update_numeric_input` *Change numeric input value and settings*

Description

Change numeric input value and settings

Usage

```
update_numeric_input(
  session,
  input_id,
  label = NULL,
  value = NULL,
  min = NULL,
  max = NULL,
  step = NULL
)

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.
input_id	The id of the input object.
label	The label to set for the input object.
value	The value to set for the input object.
min	Minimum value.
max	Maximum value.
step	Step size.
inputId	the same as input_id

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- semanticPage(
  slider_input("slider_in", 5, 0, 10),
  numeric_input("input", "Numeric input:", 0)
)

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

  observeEvent(input$slider_in, {
    x <- input$slider_in

    update_numeric_input(session, "input", value = x)
  })
}

shinyApp(ui, server)
}
```

update_progress *Update progress Semantic UI component*

Description

Change the value of a progress input on the client.

Usage

```
update_progress(
  session,
  input_id,
  type = c("increment", "decrement", "label", "value"),
  value = 1
)
```

Arguments

session	The session object passed to function given to shinyServer.
input_id	The id of the input object
type	Whether you want to increase the progress bar ("increment"), decrease the progress bar ("decrement"), update the label "label", or set it to a specific value ("value")
value	The value to increase/decrease by, or the value to be set to

update_rating_input *Update rating*

Description

Change the value of a rating input on the client. Check rating_input to learn more.

Usage

```
update_rating_input(session, input_id, label = NULL, value = NULL)
```

Arguments

session	shiny object with session info
input_id	rating input name
label	character with updated label
value	new rating value

Examples

```
## Only run examples in interactive R sessions
if (interactive()) {
library(shiny)
library(shiny.semantic)

ui <- shinyUI(
  semanticPage(
    rating_input("rate", "How do you like it?", max = 5,
                 icon = "heart", color = "yellow"),
    numeric_input("numeric_in", "", 0, min = 0, max = 5)
  )
)
server <- function(session, input, output) {
  observeEvent(input$numeric_in, {
    x <- input$numeric_in
    update_rating_input(session, "rate", value = x)
  }
  )
}
shinyApp(ui = ui, server = server)
}
```

`update_slider` *Update slider Semantic UI component*

Description

Change the value of a `slider_input` input on the client.

Usage

```
update_slider(session, input_id, value)

update_range_input(session, input_id, value, value2)

updateSliderInput(session, inputId, value, ...)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`input_id`	The id of the input object
`value`	The value to be selected for the sldier (lower value if using range).
`value2`	The upper value of the range.
`inputId`	Input name.
`...`	additional arguments

See Also

slider_input

Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  shinyApp(
    ui = semanticPage(
      p("The first slider controls the second"),
      slider_input("control", "Controller:", min = 0, max = 20, value = 10,
                   step = 1),
      slider_input("receive", "Receiver:", min = 0, max = 20, value = 10,
                   step = 1)
    ),
    server = function(input, output, session) {
      observe({
        update_slider(session, "receive", value = input$control)
      })
    }
  )
}
```

---

update_tabset *Change the selected tab of a tabset on the client*

Description

Change the selected tab of a tabset on the client

Usage

```
update_tabset(session, input_id, selected = NULL)
```

Arguments

session	The session object passed to function given to shinyServer.
input_id	The id of the tabset object.
selected	The id of the tab to be selected.

Examples

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

 ui <- semanticPage(
```

```
    actionButton("changetab", "Select Second Tab"),
    tabset(
       tabs = list(
           list(menu = "First Tab", content = "First Tab", id= "first_tab"),
           list(menu = "Second Tab", content = "Second Tab", id = "second_tab")
       ),
       active = "first_tab",
       id = "exampletabset"
    )
  )

  server <- function(input, output, session) {
      observeEvent(input$changetab,{
          update_tabset(session, "exampletabset", "second_tab")
      })
  }

  shinyApp(ui, server)
}
```

`vertical_layout` *Vertical layout*

Description

Lays out elements vertically, one by one below one another.

Usage

```
vertical_layout(
  ...,
  rows_heights = NULL,
  cell_args = "",
  adjusted_to_page = TRUE
)

verticalLayout(..., fluid = NULL)
```

Arguments

`...`	Unnamed arguments will become child elements of the layout.
`rows_heights`	Character or numeric vector indicating the widths of the individual cells. Recycling will be used if needed.
`cell_args`	character with additional attributes that should be used for each cell of the layout.
`adjusted_to_page`	if TRUE it adjust elements position in equal spaces to the size of the page
`fluid`	not supported yet (here for consistency with `shiny`)

Value

vertical layout grid object

Examples

```
if (interactive()) {
  ui <- semanticPage(
    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) { })
}
if (interactive()) {
  ui <- semanticPage(
    vertical_layout(h1("Title"), h4("Subtitle"), p("paragraph"), h3("footer"))
  )
  shinyApp(ui, server = function(input, output) { })
}
```

---

with_progress *Reporting progress (functional API)*

Description

Reports progress to the user during long-running operations.

Usage

```
with_progress(
  expr,
  min = 0,
  max = 1,
  value = min + (max - min) * 0.1,
  message = NULL,
  session = getDefaultReactiveDomain(),
  env = parent.frame(),
  quoted = FALSE
)

withProgress(
  expr,
  min = 0,
  max = 1,
  value = min + (max - min) * 0.1,
  message = NULL,
```

```
  session = getDefaultReactiveDomain(),
  env = parent.frame(),
  quoted = FALSE,
  ...
)

setProgress(
  value = NULL,
  message = NULL,
  session = getDefaultReactiveDomain(),
  ...
)

set_progress(
  value = NULL,
  message = NULL,
  session = getDefaultReactiveDomain()
)

incProgress(
  amount = 0.1,
  message = NULL,
  session = getDefaultReactiveDomain(),
  ...
)

inc_progress(
  amount = 0.1,
  message = NULL,
  session = getDefaultReactiveDomain(),
  ...
)
```

Arguments

expr	The work to be done. This expression should contain calls to ‘set_progress‘.
min	The value that represents the starting point of the progress bar. Must be less tham ‘max‘. Default is 0.
max	The value that represents the end of the progress bar. Must be greater than ‘min‘. Default is 1.
value	Single-element numeric vector; the value at which to set the progress bar, relative to ‘min‘ and ‘max‘.
message	A single-element character vector; the message to be displayed to the user, or ‘NULL‘ to hide the current message (if any).
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).
...	Arguments that may have been used in ‘shiny::withProgress‘
amount	For ‘inc_progress‘, the amount to increment the status bar. Default is 0.1.

Details

This package exposes two distinct programming APIs for working with progress. Using ‘with_progress‘ with ‘inc_progress‘ or ‘set_progress‘ provide a simple function-based interface, while the [Progress()] reference class provides an object-oriented API.

Use ‘with_progress‘ 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 ‘inc_progress‘ or ‘set_progress‘ are called. When ‘with_progress‘ exits, the corresponding progress panel will be removed.

The ‘inc_progress‘ function increments the status bar by a specified amount, whereas the ‘set_progress‘ function sets it to a specific value, and can also set the text displayed.

Generally, ‘with_progress‘/‘inc_progress‘/‘set_progress‘ 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.

When migrating from shiny applications, the functions ‘withProgress‘, ‘incProgress‘ and ‘setProgress‘ are aliases for ‘with_progress‘, ‘inc_progress‘ and ‘set_progress‘.

See Also

[Progress()]

Examples

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

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

server <- function(input, output) {
  output$plot <- renderPlot({
    with_progress(message = 'Calculation in progress',
                  detail = 'This may take a while...', value = 0, {
      for (i in 1:15) {
        inc_progress(1/15)
        Sys.sleep(0.25)
      }
    })
    plot(cars)
  })
}

shinyApp(ui, server)
}
```

Index