

# Package ‘shiny.semantic’

January 10, 2021

**Type** Package

**Title** Semantic UI Support for Shiny

**Version** 0.4.2

**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 Semantic UI - <https://fomantic-ui.com/>. It also supports universal UI input binding that works with various DOM elements.

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

**Encoding** UTF-8

**LazyData** TRUE

**License** MIT + file LICENSE

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

**Suggests** dplyr, tibble, knitr, testthat, lintr, DT, covr, leaflet, plotly

**RoxygenNote** 7.1.1

**NeedsCompilation** no

**Author** Filip Stachura [aut],  
Dominik Krzeminski [cre],  
Krystian Igras [aut],  
Adam Forys [aut],  
Paweł Przytuła [aut],  
Jakub Chojna [aut],  
Olga Mierzwa-Sulima [aut],  
Ashley Baldry [ctb],  
Jakub Chojna [ctb],  
Olga Mierzwa-Sulima [ctb],  
Pedro Manuel Coutinho da Silva [ctb],  
Paweł Przytuła [ctb],

Kamil Żyła [ctb],  
 Rabii Bouhestine [ctb],  
 Appsilon Sp. z o.o. [cph]

**Maintainer** Dominik Krzeminski <dominik@appsilon.com>

**Repository** CRAN

**Date/Publication** 2021-01-10 17:10:08 UTC

## R topics documented:

.onLoad	4
accordion	4
action_button	5
attach_rule	6
button	7
calendar	7
card	9
cards	10
checkbox_input	11
check_proper_color	12
check_semantic_theme	13
check_shiny_param	13
COLOR_PALETTE	14
counter_button	14
create_modal	15
data_frame_to_css_grid_template_areas	16
date_input	16
define_selection_type	18
digits2words	19
display_grid	19
dropdown_input	20
dropdown_menu	21
extract_icon_name	22
field	22
fields	23
file_input	24
flow_layout	26
form	27
generate_random_id	28
get_cdn_path	28
get_default_semantic_theme	29
get_dependencies	29
get_numeric	30
grid	30
grid_container_css	32
grid_template	33
header	34
horizontal_menu	35

icon . . . . .	36
label . . . . .	37
list_container . . . . .	38
list_element . . . . .	39
list_of_area_tags . . . . .	39
menu . . . . .	40
menu_divider . . . . .	41
menu_header . . . . .	42
menu_item . . . . .	42
message_box . . . . .	43
modal . . . . .	44
multiple_checkbox . . . . .	47
numeric_input . . . . .	49
parse_val . . . . .	50
Progress . . . . .	51
progress . . . . .	53
rating_input . . . . .	55
register_search . . . . .	56
render_menu_link . . . . .	57
search_field . . . . .	58
search_selection_api . . . . .	59
search_selection_choices . . . . .	60
segment . . . . .	61
selectInput . . . . .	62
semanticPage . . . . .	64
semantic_DT . . . . .	65
semantic_DTOutput . . . . .	66
set_tab_id . . . . .	66
shiny.semantic . . . . .	67
shiny_input . . . . .	67
shiny_text_input . . . . .	68
show_modal . . . . .	69
sidebar_panel . . . . .	69
SIZE_LEVELS . . . . .	71
slider_input . . . . .	72
split_args . . . . .	74
split_layout . . . . .	75
SUPPORTED_THEMES . . . . .	76
tabset . . . . .	76
textAreaInput . . . . .	78
text_input . . . . .	79
theme_selector . . . . .	80
toast . . . . .	81
uiinput . . . . .	84
uirender . . . . .	85
updateSelectInput . . . . .	85
update_action_button . . . . .	86
update_dropdown_input . . . . .	88

update_numeric_input . . . . .	89
update_progress . . . . .	90
update_rating_input . . . . .	91
update_slider . . . . .	92
update_tabset . . . . .	93
vertical_layout . . . . .	94
warn_unsupported_args . . . . .	95
with_progress . . . . .	95
%:::% . . . . .	98

**Index** **99**

`.onLoad` *Internal function that expose javascript bindings to Shiny app.*

**Description**

Internal function that expose javascript bindings to Shiny app.

**Usage**

```
.onLoad(libname, pkgname)
```

**Arguments**

<code>libname</code>	library name
<code>pkgname</code>	package name

`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 <a href="#">icon</a> 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)
}

```

---

attach_rule	<i>Internal function that creates the rule for a specific setting or behavior of the modal.</i>
-------------	---

---

**Description**

Internal function that creates the rule for a specific setting or behavior of the modal.

**Usage**

```
attach_rule(id, behavior, target, value)
```

**Arguments**

id	ID of the target modal.
behavior	What behavior is being set i. e. setting or attach events.
target	First argument of the behavior. Usually a target or a setting name.
value	Second argument of the behavior. usually an action or a setting value.

---

button	<i>Create Semantic UI Button</i>
--------	----------------------------------

---

**Description**

Create Semantic UI Button

**Usage**

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

**Arguments**

<code>input_id</code>	The input slot that will be used to access the value.
<code>label</code>	The contents of the button or link
<code>icon</code>	An optional <code>icon()</code> to appear on the button.
<code>class</code>	An optional attribute to be added to the button's class. If used parameters like <code>color</code> , <code>size</code> are ignored.
<code>...</code>	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	<i>Create Semantic UI Calendar</i>
----------	------------------------------------

---

**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**

<code>input_id</code>	ID of the calendar that will be updated
<code>value</code>	Initial value of the numeric input.
<code>placeholder</code>	Text visible in the input when nothing is inputted.
<code>type</code>	Select from 'year', 'month', 'date' and 'time'
<code>min</code>	Minimum allowed value.
<code>max</code>	Maximum allowed value.
<code>session</code>	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")
```

---

check\_semantic\_theme    *Semantic theme path validator*

---

**Description**

Semantic theme path validator

**Usage**

```
check_semantic_theme(theme_css, full_url = TRUE)
```

**Arguments**

theme_css	it can be either NULL, character with css path, or theme name
full_url	boolean flag that defines what is returned, either filename, or full path. Default TRUE

**Value**

path to theme or filename

**Examples**

```
check_semantic_theme(NULL)
check_semantic_theme("darkly")
check_semantic_theme("darkly", full_url = FALSE)
```

---

check\_shiny\_param    *Checks whether argument included as shiny exclusive parameter*

---

**Description**

A quick function to check a shiny.semantic wrapper of a shiny function to see whether any extra arguments are called that aren't required for the shiny.semantic version

**Usage**

```
check_shiny_param(name, func, ...)
```

**Arguments**

name	Function argument name
func	Name of the function in the
...	Arguments passed to the shiny.semantic version of the shiny function

**Value**

If the shiny exclusive argument is called in a shiny.semantic, then a message is posted in the UI

---

COLOR_PALETTE	<i>Semantic colors</i>
---------------	------------------------

---

**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	<i>Counter Button</i>
----------------	-----------------------

---

**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 <code>icon()</code> 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	<i>Allows for the creation of modals in the server side without being tied to a specific HTML element.</i>
--------------	--

---

**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

---

data\_frame\_to\_css\_grid\_template\_areas

*Generate CSS string representing grid template areas.*

---

### Description

Generate CSS string representing grid template areas.

### Usage

```
data_frame_to_css_grid_template_areas(areas_dataframe)
```

### Arguments

areas\_dataframe

data.frame of character representing grid areas

### Details

This is a helper function used in grid\_template()

```
areas_dataframe <- rbind(  
  c("header", "header", "header"),  
  c("menu", "main", "right1"),  
  c("menu", "main", "right2")  
)
```

```
result == "'header header header' 'menu main right1' 'menu main right2'"
```

### Value

character

---

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)
}

```

---

define\_selection\_type *Define search type if multiple*

---

### Description

Define search type if multiple

### Usage

```
define_selection_type(input_id, multiple)
```

### Arguments

input_id	character with name
multiple	multiple flag

---

digits2words	<i>Helper function that transforms digits to words</i>
--------------	--

---

**Description**

Helper function that transforms digits to words

**Usage**

```
digits2words(number)
```

**Arguments**

number            numeric digits from 1 to 10

**Value**

character with number word

---

display_grid	<i>Display grid template in a browser for easy debugging</i>
--------------	--

---

**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](https://www.w3schools.com/css/css_grid.asp)

---

dropdown_input	<i>Create dropdown Semantic UI component</i>
----------------	--

---

### 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

<code>input_id</code>	Input name. Reactive value is available under <code>input[[input_id]]</code> .
<code>choices</code>	All available options one can select from.
<code>choices_value</code>	What reactive value should be used for corresponding choice.
<code>default_text</code>	Text to be visible on dropdown when nothing is selected.
<code>value</code>	Pass value if you want to initialize selection for dropdown.
<code>type</code>	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	<i>Create Semantic UI Dropdown</i>
---------------	------------------------------------

---

## 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 <a href="https://semantic-ui.com/modules/dropdown.html">https://semantic-ui.com/modules/dropdown.html</a> 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 <a href="https://semantic-ui.com/modules/dropdown.html#/settings">https://semantic-ui.com/modules/dropdown.html#/settings</a> 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)
}

```

---

extract_icon_name	<i>Extract icon name</i>
-------------------	--------------------------

---

**Description**

Extract icon name

**Usage**

```
extract_icon_name(icon)
```

**Arguments**

icon	icon object
------	-------------

**Value**

character with icon name

---

field	<i>Create Semantic UI field tag</i>
-------	-------------------------------------

---

**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 <code>input[[input_id]]</code> .
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 <code>fileInput</code> to <code>file_input</code> like <code>type</code> .
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)
}

```

---

generate_random_id	<i>Some elements require input id, but this does not need to be specified by the user. Thus we assign random value with prefix where needed.</i>
--------------------	--

---

### Description

Some elements require input id, but this does not need to be specified by the user. Thus we assign random value with prefix where needed.

### Usage

```
generate_random_id(prefix, id_length = 20)
```

### Arguments

prefix	character with prefix add to id
id_length	numeric with length of id (default 20)

---

get_cdn_path	<i>Get CDN path semantic dependencies</i>
--------------	---

---

### Description

Internal function that returns path string from ‘shiny.custom.semantic.cdn’ options.

### Usage

```
get_cdn_path()
```

**Value**

CDN path of semantic dependencies

**Examples**

```
## Load shiny.semantic dependencies from local domain.  
options("shiny.custom.semantic.cdn" = "shiny.semantic")
```

---

get\_default\_semantic\_theme  
*Get default semantic css*

---

**Description**

Get default semantic css

**Usage**

```
get_default_semantic_theme(full_url = TRUE)
```

**Arguments**

full\_url            define return output filename or full path. Default TRUE

**Value**

path to default css semantic file or default filename

---

get\_dependencies        *Add dashboard dependencies to html*

---

**Description**

Internal function that adds dashboard dependencies to html.

**Usage**

```
get_dependencies(theme = NULL)
```

**Arguments**

theme                define theme

**Value**

Content with appended dependencies.

---

get_numeric	<i>Extracts numeric values</i>
-------------	--------------------------------

---

**Description**

Extracts numeric values

**Usage**

```
get_numeric(value)
```

**Arguments**

value	Value to be converted to numeric
-------	----------------------------------

**Value**

Numeric value

---

grid	<i>Use CSS grid template in Shiny UI</i>
------	--

---

**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\_container\_css      *Generate template string representing CSS styles of grid container div.*

---

### Description

Generate template string representing CSS styles of grid container div.

### Usage

```
grid_container_css(css_grid_template_areas, rows_height, cols_width)
```

### Arguments

css_grid_template_areas	character, CSS value for grid-template-areas
rows_height	vector of character
cols_width	vector of character

### Details

This is a helper function used in grid\_template()

```

grid_container_css(
  "'a a a' 'b b b'",
  c("50%", "50%"),
  c("100px", "2fr", "1fr")
)

```

returns

```

"display: grid;
height: 100%;
grid-template-rows: 50% 50%;
grid-template-columns: 100px 2fr 1fr;
grid-template-areas: 'a a a' 'b b b';
{{ custom_style_grid_container }}"

```

### Value

character



---

grid_template	<i>Define a template of a CSS grid</i>
---------------	--

---

**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	<i>Horizontal menu</i>
-----------------	------------------------

---

## 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

<code>class</code>	A name of an icon. Look at <a href="http://semantic-ui.com/elements/icon.html">http://semantic-ui.com/elements/icon.html</a> for all possibilities.
<code>...</code>	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 <a href="https://semantic-ui.com/elements/label.html">https://semantic-ui.com/elements/label.html</a> 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)
}
```

---

list_element	<i>Helper function to render list element</i>
--------------	---

---

### Description

Helper function to render list element

### Usage

```
list_element(header = NULL, description = NULL, icon_name = NULL)
```

### Arguments

header	character with header element
description	character with content of the list
icon_name	character with optional icon

---

list_of_area_tags	<i>Generate list of HTML div elements representing grid areas.</i>
-------------------	--

---

### Description

Generate list of HTML div elements representing grid areas.

### Usage

```
list_of_area_tags(area_names)
```

**Arguments**

area\_names      vector with area names

**Details**

This is a helper function used in `grid_template()`

```
list_of_area_tags(c("header", "main", "footer"))
```

returns the following list:

```
[[1]] <div id="{{ grid_id }}-header" style="grid-area: header; {{ header_custom_css }}">{{ header }}</div>
[[2]] <div id="{{ grid_id }}-main" style="grid-area: main; {{ main_custom_css }}">{{ main }}</div>
[[3]] <div id="{{ grid_id }}-footer" style="grid-area: footer; {{ footer_custom_css }}">{{ footer }}</div>
```

**Value**

list of `shiny::tags$div`

---

menu

*Create Semantic UI Menu*

---

**Description**

This creates a menu using Semantic UI.

**Usage**

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

**Arguments**

...	Menu items to be created. Use <code>menu_item</code> function to create new menu item. Use <code>dropdown_menu(is_menu_item = TRUE, ...)</code> function to create new dropdown menu item. Use <code>menu_header</code> and <code>menu_divider</code> functions to customize menu format.
class	Class extension. Look at <a href="https://semantic-ui.com/collections/menu.html">https://semantic-ui.com/collections/menu.html</a> 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	<i>Create Semantic UI Header Item</i>
-------------	---------------------------------------

---

**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	<i>Create Semantic UI Menu Item</i>
-----------	-------------------------------------

---

**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 parameter defines its href attribute.

**See Also**

menu

---

message_box	<i>Create Semantic UI Message box</i>
-------------	---------------------------------------

---

**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 <a href="https://semantic-ui.com/collections/message.html">https://semantic-ui.com/collections/message.html</a> for all possibilities.
icon_name	If the message is of the type 'icon', specify the icon. Look at <a href="http://semantic-ui.com/elements/icon.html">http://semantic-ui.com/elements/icon.html</a> 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)
}
if (interactive()) {

```

```
library(shiny)
library(shiny.semantic)
shinyApp(
  ui = semanticPage(
    actionButton("show", "Show modal dialog")
  ),
  server = function(input, output) {
    observeEvent(input$show, {
      showModal(modalDialog(
        title = "Important message",
        "This modal will close after 3 sec.", easyClose = FALSE
      ))
      Sys.sleep(3)
      removeModal()
    })
  }
)
```

---

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], value = "A"),
        p("Selected letters:"),
        textOutput("selected_letters"),
        tags$br(),
        h1("Radioboxes"),
        multiple_radio("radioboxes", "Select Letter", LETTERS[1:6], value = "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,  
  min = NA,  
  max = NA,  
  step = NA,  
  type = NULL,  
  icon = NULL,  
  placeholder = NULL,  
  ...  
)  
  
numericInput(  
  inputId,  
  label,  
  value,  
  min = NA,  
  max = NA,  
  step = NA,  
  width = NULL,  
  ...  
)
```

### Arguments

<code>input_id</code>	Input name. Reactive value is available under <code>input[[input_id]]</code> .
<code>label</code>	Display label for the control, or <code>NULL</code> for no label.
<code>value</code>	Initial value of the numeric input.
<code>min</code>	Minimum allowed value.
<code>max</code>	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 <code>numeric_input</code> like <code>type</code> or <code>icon</code> .
inputId	The input slot that will be used to access the value.
width	The width of the input.

### Details

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)
}
```

---

parse\_val

*Parse the 'shiny\_input' value from JSON*

---

### Description

Parse the 'shiny\_input' value from JSON

### Usage

```
parse_val(val)
```

### Arguments

val                    value to get from JSON

### Value

Value of type defined in 'shiny\_input'

## 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

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

---

## 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 <code>icon()</code> 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

<code>session</code>	Shiny server session
<code>data</code>	Data (the value that is passed into <code>registerDataObj</code> )
<code>search_query</code>	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	<i>Render menu link</i>
------------------	-------------------------

---

## 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	<i>Create search field Semantic UI component</i>
--------------	--

---

**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 <code>input[[input_id]]</code> .
search_api_url	Register custom API url with server JSON Response containing fields ‘title’ and ‘description’.
default_text	Text to be visible on search 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. <pre>#' @examples ## Only run examples in interactive R sessions if (interactive()) library(shiny) library(shiny.semantic) library(gapminder) library(dplyr) ui &lt;- function() shinyUI( semanticPage( title = "Dropdown example", uiOutput("search_letters"), p("Selected letter:"), textOutput("selected_letters") ) ) server &lt;- shinyServer(function(input, output, session)   search_api &lt;- function(gapminder, q) has_matching &lt;- function(field) startsWith(field, q)   gapminder %&gt;% mutate(country = as.character(country)) %&gt;% select(country) %&gt;% unique %&gt;% filter(has_matching(country)) %&gt;% head(5) %&gt;% transmute(name = country, value = country)   search_api_url &lt;- shiny.semantic::register_search(session, gapminder, search_api)   output\$search_letters &lt;- shiny::renderUI( search_selection_api("search_result", search_api_url, multiple = TRUE) )   output\$selected_letters &lt;- renderText(input[["search_result"]]) ) shinyApp(ui = ui(), server = server)</pre>

---

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**

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

**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 <code>multiple = TRUE</code> ). 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 <a href="#">dropdown_input</a> .

### 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 SUPPORTED_THEMES or at <a href="http://semantic-ui-forest.com/themes">http://semantic-ui-forest.com/themes</a> .
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 semantic 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())
```

### Arguments

...                    datatable parameters, check `?DT::datatable` to learn more.  
options                datatable options, 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	<i>Semantic DT Output</i>
-------------------	---------------------------

---

**Description**

Semantic DT Output

**Usage**

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

**Arguments**

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

**Value**

DT Output with semantic style

---

set_tab_id	<i>Sets tab id if not provided</i>
------------	------------------------------------

---

**Description**

Sets tab id if it wasn't provided

**Usage**

```
set_tab_id(tab)
```

**Arguments**

tab            A tab. Tab is a list of three elements - first element defines menu item, second element defines tab content, third optional element defines tab id.

---

shiny.semantic	<i>Semantic UI wrapper for Shiny</i>
----------------	--------------------------------------

---

### Description

With this library it's easy to wrap Shiny with Semantic UI components. Add a few simple lines of code and some CSS classes to give your UI a fresh, modern and highly interactive look.

### Options

There are a number of global options that affect shiny.semantic as well as Shiny behavior. The options can be set globally with 'options()'

**shiny.custom.semantic.cdn (defaults is internal CDN)** This controls from where the css and javascripts will be downloaded.

**shiny.semantic.local (defaults to 'FALSE')** This allows to use only local dependency.

**shiny.custom.semantic (defaults to 'NULL')** This allows to set custom local path to semantic dependencies.

**shiny.minified (defaults to 'TRUE')** Defines including JavaScript as a minified or un-minified file.

---

shiny_input	<i>Create universal Shiny input binding</i>
-------------	---

---

### 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 <code>input[[input_id]]</code> .
shiny_ui	UI of HTML component presenting this input to the users. This UI should allow to extract its value with jQuery <code>\$.val()</code> function.
value	An optional argument with value that should be set for this input. Can be used to store persistent input values 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())
```

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

```
remove_all_modals(session = shiny::getDefaultReactiveDomain())
```

```
removeModal(session = shiny::getDefaultReactiveDomain())
```

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

### Arguments

`id` ID of the modal that will be displayed.

`session` The session object passed to function given to shinyServer.

### 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**

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

```

---

 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

<code>input_id</code>	Input name. Reactive value is available under <code>input[[input_id]]</code> .
<code>value</code>	The initial value to be selected for the slider (lower value if using range).
<code>min</code>	The minimum value allowed to be selected for the slider.
<code>max</code>	The maximum value allowed to be selected for the slider.
<code>step</code>	The interval between each selectable value of the slider.
<code>class</code>	UI class of the slider. Can include "labeled" and "ticked".
<code>custom_ticks</code>	A vector of custom labels to be added to the slider. Will ignore min and max
<code>inputId</code>	Input name.
<code>label</code>	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\_args

*Split arguments to positional and named*

---

## Description

Split arguments to positional and named

## Usage

```
split_args(...)
```

## Arguments

... arguments to split

## Value

A list with two named elements:

- positional, a list of the positional arguments,
- named, a list of the named arguments.

---

split_layout	<i>Split layout</i>
--------------	---------------------

---

**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)
}

```

---

SUPPORTED_THEMES	<i>Supported semantic themes</i>
------------------	----------------------------------

---

### Description

Supported semantic themes

### Usage

SUPPORTED\_THEMES

### Format

An object of class character of length 17.

---

tabset	<i>Create Semantic UI tabs</i>
--------	--------------------------------

---

### 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**

<code>tabs</code>	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.
<code>active</code>	Id of the active tab. If NULL first tab will be active.
<code>id</code>	Id of the menu element (default: randomly generated id)
<code>menu_class</code>	Class for the menu element (default: "top attached tabular")
<code>tab_content_class</code>	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 = "examplatabset"
  ),
  h2("Active Tab:"),
  textOutput("activetab")
)
server <- function(input, output) {
  output$activetab <- renderText(input$examplatabset)
}
```

```
    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

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

### Examples

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

---

text_input	<i>Create Semantic UI Text Input</i>
------------	--------------------------------------

---

**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**

<code>input_id</code>	Input name. Reactive value is available under <code>input[[input_id]]</code> .
<code>label</code>	character with label put on the left from the input
<code>value</code>	Pass value if you want to have default text.
<code>type</code>	Change depending what type of input is wanted. See details for options.
<code>placeholder</code>	Text visible in the input when nothing is inputted.
<code>attribs</code>	A named list of attributes to assign to the input.
<code>inputId</code>	Input name. The same as <code>input_id</code> .
<code>width</code>	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	<i>Themes changer dropdown</i>
----------------	--------------------------------

---

### Description

Themes changer dropdown

### Usage

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

### Arguments

input_id	Id of dropdown. <code>input[[input_id]]</code> 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. Default 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)

```

---

`uiinput`*Create Semantic UI Input*

---

**Description**

This creates an input shell for the actual input

**Usage**

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

**Arguments**

<code>...</code>	Other arguments to be added as attributes of the tag (e.g. style, class or childrens etc.)
<code>class</code>	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	<i>Render semanticui htmlwidget</i>
----------	-------------------------------------

---

**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	<i>Change the value of a select input on the client</i>
-------------------	---

---

**Description**

Update selectInput 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)
}
```

**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

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

## Examples

```
if (interactive()) {

  library(shiny)
  library(shiny.semantic)

  ui <- function() {
    shinyUI(
      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 <- shinyServer(function(input, output, session) {
    output$selected_letter <- renderText(paste(input[["simple_dropdown"]], collapse = ", "))
  })
}
```



```
    observeEvent(input$simple_button, {
      update_dropdown(session, "simple_dropdown", value = "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,
  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	<i>Update progress Semantic UI component</i>
-----------------	--

---

### 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 slider (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	<i>Change the selected tab of a tabset on the client</i>
---------------	--

---

**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 = "examplatabset"
    )
  )

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

  shinyApp(ui, server)
}
```

---

vertical_layout	<i>Vertical layout</i>
-----------------	------------------------

---

### 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) { })
}

```

---

warn\_unsupported\_args *Warn that there are not supported arguments*

---

### Description

This throws warning if there are parameters not supported by semantic.

### Usage

```
warn_unsupported_args(args)
```

### Arguments

args                    list or vector with extra arguments

---

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

<code>expr</code>	The work to be done. This expression should contain calls to <code>'set_progress'</code> .
<code>min</code>	The value that represents the starting point of the progress bar. Must be less than <code>'max'</code> . Default is 0.
<code>max</code>	The value that represents the end of the progress bar. Must be greater than <code>'min'</code> . Default is 1.
<code>value</code>	Single-element numeric vector; the value at which to set the progress bar, relative to <code>'min'</code> and <code>'max'</code> .
<code>message</code>	A single-element character vector; the message to be displayed to the user, or <code>'NULL'</code> to hide the current message (if any).
<code>session</code>	The Shiny session object, as provided by <code>'shinyServer'</code> 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

- \* **datasets**
  - COLOR\_PALETTE, [14](#)
  - SIZE\_LEVELS, [71](#)
  - SUPPORTED\_THEMES, [76](#)
- .onLoad, [4](#)
- %::%, [98](#)
  
- a (checkbox\_input), [11](#)
- accordion, [4](#)
- action\_button, [5](#)
- actionButton (action\_button), [5](#)
- attach\_rule, [6](#)
  
- button, [7](#)
  
- calendar, [7](#)
- card, [9](#)
- cards, [10](#)
- check\_proper\_color, [12](#)
- check\_semantic\_theme, [13](#)
- check\_shiny\_param, [13](#)
- checkbox (checkbox\_input), [11](#)
- checkbox\_input, [11](#)
- checkboxInput (checkbox\_input), [11](#)
- close\_toast (toast), [81](#)
- COLOR\_PALETTE, [14](#)
- counter\_button, [14](#)
- create\_modal, [15](#)
- creates (checkbox\_input), [11](#)
  
- data\_frame\_to\_css\_grid\_template\_areas,  
    [16](#)
- date\_input, [16](#)
- dateInput (date\_input), [16](#)
- define\_selection\_type, [18](#)
- digits2words, [19](#)
- display\_grid, [19](#)
- dropdown\_input, [20](#), [63](#), [88](#)
- dropdown\_menu, [21](#)
  
- extract\_icon\_name, [22](#)
  
- field, [22](#)
- fields, [23](#)
- file\_input, [24](#)
- fileInput (file\_input), [24](#)
- flow\_layout, [26](#)
- flowLayout (flow\_layout), [26](#)
- form, [27](#)
  
- generate\_random\_id, [28](#)
- get\_cdn\_path, [28](#)
- get\_default\_semantic\_theme, [29](#)
- get\_dependencies, [29](#)
- get\_numeric, [30](#)
- grid, [30](#)
- grid\_container\_css, [32](#)
- grid\_template, [33](#)
  
- header, [34](#)
- hide\_modal (show\_modal), [69](#)
- horizontal\_menu, [35](#)
  
- icon, [6](#), [7](#), [14](#), [36](#), [55](#)
- inc\_progress (with\_progress), [95](#)
- incProgress (with\_progress), [95](#)
  
- label, [37](#)
- list\_container, [38](#)
- list\_element, [39](#)
- list\_of\_area\_tags, [39](#)
  
- main\_panel (sidebar\_panel), [69](#)
- mainPanel (sidebar\_panel), [69](#)
- menu, [40](#)
- menu\_divider, [41](#)
- menu\_header, [42](#)
- menu\_item, [42](#)
- message\_box, [43](#)
- modal, [44](#)
- modalDialog (modal), [44](#)
- multiple\_checkbox, [47](#)
- multiple\_radio (multiple\_checkbox), [47](#)

numeric\_input, [49, 50](#)  
 numericInput (numeric\_input), [49](#)  
  
 parse\_val, [50](#)  
 Progress, [51](#)  
 progress, [53, 90](#)  
  
 range\_input (slider\_input), [72](#)  
 rating\_input, [55](#)  
 register\_search, [56](#)  
 remove\_all\_modals (show\_modal), [69](#)  
 remove\_modal (show\_modal), [69](#)  
 removeModal (show\_modal), [69](#)  
 removeNotification (toast), [81](#)  
 render\_menu\_link, [57](#)  
  
 search\_field, [58](#)  
 search\_selection\_api, [59](#)  
 search\_selection\_choices, [60](#)  
 segment, [61](#)  
 selectInput, [62](#)  
 Semantic (checkbox\_input), [11](#)  
 semantic\_DT, [65](#)  
 semantic\_DTOutput, [66](#)  
 semanticPage, [64](#)  
 set\_progress (with\_progress), [95](#)  
 set\_tab\_id, [66](#)  
 setProgress (with\_progress), [95](#)  
 shiny.semantic, [67](#)  
 shiny\_input, [67](#)  
 shiny\_text\_input, [68](#)  
 show\_modal, [69](#)  
 showModal (create\_modal), [15](#)  
 showNotification (toast), [81](#)  
 sidebar\_layout (sidebar\_panel), [69](#)  
 sidebar\_panel, [69](#)  
 sidebarLayout (sidebar\_panel), [69](#)  
 sidebarPanel (sidebar\_panel), [69](#)  
 SIZE\_LEVELS, [71](#)  
 slider\_input, [72, 92](#)  
 sliderInput (slider\_input), [72](#)  
 split\_args, [74](#)  
 split\_layout, [75](#)  
 splitLayout (split\_layout), [75](#)  
 styles. (checkbox\_input), [11](#)  
 SUPPORTED\_THEMES, [76](#)  
  
 tabset, [76](#)  
 text\_input, [79](#)  
  
 textAreaInput, [78](#)  
 textInput (text\_input), [79](#)  
 theme\_selector, [80](#)  
 This (checkbox\_input), [11](#)  
 toast, [81](#)  
 toggle (checkbox\_input), [11](#)  
  
 UI (checkbox\_input), [11](#)  
 uiinput, [84](#)  
 uirender, [85](#)  
 update\_action\_button, [86](#)  
 update\_calendar (calendar), [7](#)  
 update\_dropdown\_input, [88](#)  
 update\_numeric\_input, [89](#)  
 update\_progress, [90](#)  
 update\_range\_input (update\_slider), [92](#)  
 update\_rating\_input, [91](#)  
 update\_slider, [73, 92](#)  
 update\_tabset, [93](#)  
 updateActionButton  
     (update\_action\_button), [86](#)  
 updateCheckboxInput, [11](#)  
 updateNumericInput, [50](#)  
 updateNumericInput  
     (update\_numeric\_input), [89](#)  
 updateSelectInput, [85](#)  
 updateSliderInput (update\_slider), [92](#)  
 updateTextAreaInput, [80](#)  
 updateTextInput, [80](#)  
 using (checkbox\_input), [11](#)  
  
 vertical\_layout, [94](#)  
 verticalLayout (vertical\_layout), [94](#)  
  
 warn\_unsupported\_args, [95](#)  
 with\_progress, [95](#)  
 withProgress (with\_progress), [95](#)