Package ‘bslib’

May 16, 2026

Title Custom 'Bootstrap' 'Sass' Themes for 'shiny' and 'rmarkdown'

Version 0.11.0

Description Simplifies custom 'CSS' styling of both 'shiny' and 'rmarkdown' via 'Bootstrap' 'Sass'. Supports 'Bootstrap' 3, 4 and 5 as well as their various 'Bootswatch' themes. An interactive widget is also provided for previewing themes in real time.

License MIT + file LICENSE

URL https://rstudio.github.io/bslib/, https://github.com/rstudio/bslib

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

Depends R (>= 2.10)

Imports base64enc, cachem, fastmap (>= 1.1.1), grDevices, htmltools (>= 0.5.8), jquerylib (>= 0.1.3), jsonlite, lifecycle, memoise (>= 2.0.1), mime, rlang, sass (>= 0.4.9)

Suggests brand.yml, bsicons, curl, fontawesome, future, ggplot2, knitr, lattice, magrittr, rappdirs, rmarkdown (>= 2.7), shiny (>= 1.11.1.9000), testthat, thematic, tools, utils, withr, yaml

Config/Needs/deploy BH, chiflights22, colourpicker, commonmark, cpp11, cpsievert/chiflights22, cpsievert/histoslider, dplyr, DT, ggplot2, ggridges, gt, hexbin, histoslider, htmlwidgets, lattice, leaflet, lubridate, markdown, modelr, plotly, reactable, reshape2, rprojroot, rsconnect, rstudio/shiny, scales, styler, tibble

Config/Needs/routine chromote, desc, renv

Config/Needs/website brio, crosstalk, dplyr, DT, ggplot2, glue, htmlwidgets, leaflet, lorem, palmerpenguins, plotly, purrr, rprojroot, rstudio/htmltools, scales, stringr, tidyr, webshot2

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first zzzz-bs-sass, fonts, zzz-precompile, theme-*, rmd-*

Encoding UTF-8

Collate 'accordion.R' 'breakpoints.R' 'bs-current-theme.R'
'bs-dependencies.R' 'bs-global.R' 'bs-remove.R'
'bs-theme-layers.R' 'bs-theme-preset-bootswatch.R'
'bs-theme-preset-brand.R' 'bs-theme-preset-builtin.R'
'bs-theme-preset.R' 'utils.R' 'bs-theme-preview.R'
'bs-theme-update.R' 'bs-theme.R' 'bslib-package.R' 'buttons.R'
'card.R' 'deprecated.R' 'files.R' 'fill.R' 'imports.R'
'input-code-editor.R' 'input-dark-mode.R' 'input-submit.R'
'input-switch.R' 'layout.R' 'nav-items.R' 'nav-update.R'
'navbar_options.R' 'navs-legacy.R' 'navs.R' 'onLoad.R' 'page.R'
'popover.R' 'precompiled.R' 'print.R' 'shiny-devmode.R'
'sidebar.R' 'staticimports.R' 'toast.R' 'toolbar.R' 'tooltip.R'
'utils-deps.R' 'utils-shiny.R' 'utils-tags.R' 'value-box.R'
'version-default.R' 'versions.R'

NeedsCompilation no

Author Carson Sievert [aut, cre] (ORCID:
<https://orcid.org/0000-0002-4958-2844>),
Joe Cheng [aut],
Garrick Aden-Buie [aut] (ORCID:
<https://orcid.org/0000-0002-7111-0077>),
Posit Software, PBC [cph, fnd],
Bootstrap contributors [ctb] (Bootstrap library),
Twitter, Inc [cph] (Bootstrap library),
Javi Aguilar [ctb, cph] (Bootstrap colorpicker library),
Thomas Park [ctb, cph] (Bootswatch library),
PayPal [ctb, cph] (Bootstrap accessibility plugin)

Maintainer Carson Sievert <carson@posit.co>

Repository CRAN

Date/Publication 2026-05-16 08:10:07 UTC

Contents

`accordion` *Create a vertically collapsing accordion*

Description

An accordion can be used to organize UI elements and content in a limited space. It comprises multiple, vertically stacked panels that expand or collapse when clicked, providing a compact layout that works well for grouping input elements in a `sidebar()` or for organizing detailed context-specific information.

Usage

```
accordion(
  ...,
  id = NULL,
  open = NULL,
  multiple = TRUE,
  class = NULL,
  width = NULL,
  height = NULL
)

accordion_panel(title, ..., value = title, icon = NULL)
```

Arguments

...	Named arguments become attributes on the element. Unnamed arguments should be accordion_panel()s.
id	If provided, you can use input$id in your server logic to determine which of the accordion_panel()s are currently active. The value will correspond to the accordion_panel()'s value argument.
open	A character vector of accordion_panel() values to open (i.e., show) by default. The default value of NULL will open the first accordion_panel(). Use a value of TRUE to open all (or FALSE to open none) of the items. It's only possible to open more than one panel when multiple=TRUE.
multiple	Whether multiple accordion_panel() can be open at once.
class	Additional CSS classes to include on the accordion div.
width, height	Any valid CSS unit; for example, height="100%".
title	A title to appear in the accordion_panel()'s header.
value	A character string that uniquely identifies this panel.
icon	A htmltools::tag child (e.g., bsicons::bs_icon()) which is positioned just before the title.

References

bslib's accordion component is derived from the Bootstrap Accordion component. Accordions are also featured on the bslib website:

- Get Started: Dashboards
- Sidebars: Accordions

See Also

accordion_panel_set(), accordion_panel_open() and accordion_panel_close() programmatically interact with the state of an accordion panel.

accordion_panel_insert(), accordion_panel_remove() and accordion_panel_update() add or remove accordion panels from an accordion.

Other Components: card(), popover(), tooltip(), value_box()

Examples

```
items <- lapply(LETTERS, function(x) {
  accordion_panel(paste("Section", x), paste("Some narrative for section", x))
})

# First shown by default
accordion(!!!items)
# Nothing shown by default
accordion(!!!items, open = FALSE)
# Everything shown by default
accordion(!!!items, open = TRUE)

# Show particular sections
accordion(!!!items, open = "Section B")
accordion(!!!items, open = c("Section A", "Section B"))

# Provide an id to create a shiny input binding
library(shiny)
library(bslib)

ui <- page_fluid(
  accordion(!!!items, id = "acc")
)

server <- function(input, output) {
  observe(print(input$acc))
}

shinyApp(ui, server)
```

---

accordion_panel_set *Dynamically update accordions*

Description

Dynamically update/modify `accordion()`s in a Shiny app. To be updated programmatically, the `accordion()` must have an `id`. These functions require an active Shiny session and only work with a running Shiny app.

Usage

```
accordion_panel_set(id, values, session = get_current_session())

accordion_panel_open(id, values, session = get_current_session())

accordion_panel_close(id, values, session = get_current_session())

accordion_panel_insert(
```

```
  id,
  panel,
  target = NULL,
  position = c("after", "before"),
  session = get_current_session()
)

accordion_panel_remove(id, target, session = get_current_session())

accordion_panel_update(
  id,
  target,
  ...,
  title = NULL,
  value = NULL,
  icon = NULL,
  session = get_current_session()
)
```

Arguments

id	an character string that matches an existing accordion()'s id.
values	either a character string (used to identify particular accordion_panel()s by their value) or TRUE (i.e., all values).
session	a shiny session object (the default should almost always be used).
panel	an accordion_panel().
target	The value of an existing panel to insert next to. If removing: the value of the accordion_panel() to remove.
position	Should panel be added before or after the target? When target is NULL (the default), "after" will append after the last panel and "before" will prepend before the first panel.
...	Elements that become the new content of the panel.
title	A title to appear in the accordion_panel()'s header.
value	A character string that uniquely identifies this panel.
icon	A htmltools::tag child (e.g., bsicons::bs_icon()) which is positioned just before the title.

Functions

- accordion_panel_set(): same as accordion_panel_open(), except it also closes any currently open panels.
- accordion_panel_open(): open accordion_panel()s.
- accordion_panel_close(): close accordion_panel()s.
- accordion_panel_insert(): insert a new accordion_panel()
- accordion_panel_remove(): remove accordion_panel()s.
- accordion_panel_update(): update a accordion_panel().

See Also

accordion() and accordion_panel() create the accordion component.

---

as_fill_carrier *Test and/or coerce fill behavior*

Description

Filling layouts in bslib are built on the foundation of fillable containers and fill items (fill carriers are both fillable and fill). This is why most bslib components (e.g., card(), card_body(), layout_sidebar()) possess both fillable and fill arguments (to control their fill behavior). However, sometimes it's useful to add, remove, and/or test fillable/fill properties on arbitrary htmltools::tag(), which these functions are designed to do.

Usage

```
as_fill_carrier(
  x,
  ...,
  min_height = NULL,
  max_height = NULL,
  gap = NULL,
  class = NULL,
  style = NULL,
  css_selector = NULL
)

as_fillable_container(
  x,
  ...,
  min_height = NULL,
  max_height = NULL,
  gap = NULL,
  class = NULL,
  style = NULL,
  css_selector = NULL
)

as_fill_item(
  x,
  ...,
  min_height = NULL,
  max_height = NULL,
  class = NULL,
  style = NULL,
  css_selector = NULL
```

```
)

remove_all_fill(x)

is_fill_carrier(x)

is_fillable_container(x)

is_fill_item(x)
```

Arguments

`x`	An `htmltools::tag()`.
`...`	Currently ignored.
`min_height, max_height`	Any valid CSS unit (e.g., `150`).
`gap`	Any valid CSS unit.
`class`	A character vector of class names to add to the tag.
`style`	A character vector of CSS properties to add to the tag.
`css_selector`	A character string containing a CSS selector for targeting particular (inner) tag(s) of interest. For more details on what selector(s) are supported, see `htmltools::tagAppendAttribu`

Details

Although `as_fill()`, `as_fillable()`, and `as_fill_carrier()` can work with non-tag objects that have a htmltools::as.tags method (e.g., htmlwidgets), they return the "tagified" version of that object.

Value

- For `as_fill()`, `as_fillable()`, and `as_fill_carrier()`: the *tagified* version `x`, with relevant tags modified to possess the relevant fill properties.
- For `is_fill()`, `is_fillable()`, and `is_fill_carrier()`: a logical vector, with length matching the number of top-level tags that possess the relevant fill properties.

References

The Filling Layouts article on the bslib website introduces the concept of fillable containers and fill items.

See Also

These functions provide a convenient interface to the underlying `htmltools::bindFillRole()` function.

Examples

```
library(shiny)
library(bslib)
shinyApp(
  page_fillable(
    # without `as_fill_carrier()`, the plot won't fill the page because
    # `uiOutput()` is neither a fillable container nor a fill item by default.
    as_fill_carrier(uiOutput("ui"))
  ),
  function(input, output) {
    output$ui <- renderUI({
      div(
        class = "bg-info text-white",
        as_fill_item(),
        "A fill item"
      )
    })
  }
)
```

`bind_task_button` *Bind* `input_task_button` *to* `ExtendedTask`

Description

Sets up a shiny::ExtendedTask to relay its state to an existing `input_task_button()`, so the task button stays in its "busy" state for as long as the extended task is running.

Note that `bind_task_button` does *not* automatically cause button presses to invoke the extended task; you still need to use `shiny::bindEvent()` (or `shiny::observeEvent()`) to cause the button press to trigger an invocation, as in the example below.

`bind_task_button` cannot be used to bind one task button to multiple `ExtendedTask` objects; if you attempt to do so, any bound `ExtendedTask` that completes will cause the button to return to "ready" state.

Usage

```
bind_task_button(target, task_button_id, ...)

## Default S3 method:
bind_task_button(target, task_button_id, ...)

## S3 method for class 'ExtendedTask'
bind_task_button(target, task_button_id, ..., session = get_current_session())
```

Arguments

target	The target object (i.e. ExtendedTask).
task_button_id	A string matching the id argument passed to the corresponding input_task_button() call.
...	Further arguments passed to other methods.
session	A Shiny session object (the default should almost always be used).

Value

The target object that was passed in.

Examples

```
library(shiny)
library(bslib)
library(future)
plan(multisession)

ui <- page_sidebar(
  sidebar = sidebar(
    input_task_button("recalc", "Recalculate")
  ),
  textOutput("outval")
)

server <- function(input, output) {
  rand_task <- ExtendedTask$new(function() {
    future({
      # Slow operation goes here
      Sys.sleep(2)
      runif(1)
    }, seed = TRUE)
  })

  # Make button state reflect task.
  # If using R >=4.1, you can do this instead:
  # rand_task <- ExtendedTask$new(...) |> bind_task_button("recalc")
  bind_task_button(rand_task, "recalc")

  observeEvent(input$recalc, {
    rand_task$invoke()
  })

  output$outval <- renderText({
    rand_task$result()
  })
}

shinyApp(ui, server)
```

bootswatch_themes *Obtain a list of all available bootswatch themes.*

Description

Obtain a list of all available bootswatch themes.

Usage

```
bootswatch_themes(version = version_default(), full_path = FALSE)
```

Arguments

version	The major version of Bootswatch.
full_path	Whether to return a path to the installed theme.

Value

Returns a character vector of Bootswatch themes.

See Also

Other Bootstrap theme utility functions: bs_get_variables(), builtin_themes(), theme_bootswatch(), theme_version(), versions()

breakpoints *Define breakpoint values*

Description

A generic constructor for responsive breakpoints.

Usage

```
breakpoints(..., xs = NULL, sm = NULL, md = NULL, lg = NULL)
```

Arguments

...	Other breakpoints (e.g., xl).
xs	The default value to apply to the xs breakpoint. Note that this breakpoint is generally equivalent to "all sizes" and is typically treated as the base case or a value to apply by default across all breakpoints unless overridden by a larger breakpoint.
sm	Values to apply at the sm breakpoint.
md	Values to apply at the md breakpoint.
lg	Values to apply at the lg breakpoint.

References

Bootstrap's Breakpoints article provides more detail on breakpoints and how they are used and customized.

See Also

`breakpoints()` is used by `layout_columns()`.

Examples

```
breakpoints(sm = c(4, 4, 4), md = c(3, 3, 6), lg = c(-2, 8, -2))
```

---

bs_add_variables *Add low-level theming customizations*

Description

These functions provide direct access to the layers of a bslib theme created with `bs_theme()`. Learn more about composable Sass layers on the **sass** website.

Usage

```
bs_add_variables(
  theme,
  ...,
  .where = "defaults",
  .default_flag = identical(.where, "defaults")
)

bs_add_rules(theme, rules)

bs_add_functions(theme, functions)

bs_add_mixins(theme, mixins)

bs_bundle(theme, ...)
```

Arguments

`theme`	A `bs_theme()` object.
`...`	• `bs_add_variables()`: Should be named Sass variables or values that can be passed in directly to the `defaults` argument of a `sass::sass_layer()`. • `bs_bundle()`: Should be arguments that can be handled by `sass::sass_bundle()` to be appended to the `theme`
`.where`	Whether to place the variable definitions before other Sass `"defaults"`, after other Sass `"declarations"`, or after other Sass `"rules"`.

`.default_flag`	Whether or not to add a `!default` flag (if missing) to variable expressions. It's recommended to keep this as `TRUE` when `.where = "defaults"`.
`rules`	Sass rules. Anything understood by `sass::as_sass()` may be provided (e.g., a list, character vector, `sass::sass_file()`, etc)
`functions`	A character vector or `sass::sass_file()` containing functions definitions.
`mixins`	A character vector or `sass::sass_file()` containing mixin definitions.

Details

Compared to higher-level theme customization available in `bs_theme()`, these functions are a more direct interface to Bootstrap Sass, and therefore, do nothing to ensure theme customizations are portable between major Bootstrap versions.

Value

Returns a modified `bs_theme()` object.

Functions

- `bs_add_variables()`: Add Bootstrap Sass variable defaults.
- `bs_add_rules()`: Add additional Sass rules.
- `bs_add_functions()`: Add additional Sass functions.
- `bs_add_mixins()`: Add additional Sass mixins.
- `bs_bundle()`: Add additional `sass::sass_bundle()` objects to an existing `theme`.

References

- bslib's theming capabilities are powered by the sass package.
- Learn more about composable Sass layers on the **sass** website.

See Also

`bs_theme()` creates a Bootstrap theme object, and is the best place to start learning about bslib's theming capabilities.

Other Bootstrap theme functions: `bs_current_theme()`, `bs_dependency()`, `bs_global_theme()`, `bs_remove()`, `bs_theme()`, `bs_theme_dependencies()`, `bs_theme_preview()`

Examples

```
# Function to preview the styling a (primary) Bootstrap button
library(htmltools)
button <- tags$a(class = "btn btn-primary", href = "#", role = "button", "Hello")
preview_button <- function(theme) {
  browsable(tags$body(bs_theme_dependencies(theme), button))
}

# Here we start with a theme based on a Bootswatch theme,
```

```
# then override some variable defaults
theme <- bs_add_variables(
  bs_theme(bootswatch = "sketchy", primary = "orange"),
  "body-bg" = "#EEEEEE",
  "font-family-base" = "monospace",
  "font-size-base" = "1.4rem",
  "btn-padding-y" = ".16rem",
  "btn-padding-x" = "2rem"
)

preview_button(theme)

# If you need to set a variable based on another Bootstrap variable
theme <- bs_add_variables(theme, "body-color" = "$success", .where = "declarations")
preview_button(theme)

# Start a new global theme and add some custom rules that
# use Bootstrap variables to define a custom styling for a
# 'person card'
person_rules <- system.file("custom", "person.scss", package = "bslib")
theme <- bs_add_rules(bs_theme(), sass::sass_file(person_rules))

# Include custom CSS that leverages bootstrap Sass variables
person <- function(name, title, company) {
  tags$div(
    class = "person",
    h3(class = "name", name),
    div(class = "title", title),
    div(class = "company", company)
  )
}

page_fluid(
  theme = theme,
  person("Andrew Carnegie", "Owner", "Carnegie Steel Company"),
  person("John D. Rockefeller", "Chairman", "Standard Oil")
)
```

`bs_current_theme` *Obtain the currently active theme at render time*

Description

Intended for advanced use by developers to obtain the currently active theme *at render time* and primarily for implementing themable widgets that can't otherwise be themed via `bs_dependency_defer()`.

Usage

```
bs_current_theme(session = get_current_session(FALSE))
```

Arguments

session	The current Shiny session (if any).

Details

This function should generally only be called at print/render time. For example:

- Inside the preRenderHook of htmlwidgets::createWidget().
- Inside of a custom print method that generates htmltools::tags.
- Inside of a htmltools::tagFunction()

Calling this function at print/render time is important because it does different things based on the context in which it's called:

- If a reactive context is active, session$getCurrentTheme() is called (which is a reactive read).
- If no reactive context is active, shiny::getCurrentTheme() is called (which returns the current app's theme, if relevant).
- If shiny::getCurrentTheme() comes up empty, then bs_global_get() is called, which is relevant for rmarkdown::html_document(), and possibly other static rendering contexts.

Value

Returns a bs_theme() object.

See Also

Other Bootstrap theme functions: bs_add_variables(), bs_dependency(), bs_global_theme(), bs_remove(), bs_theme(), bs_theme_dependencies(), bs_theme_preview()

---

bs_dependency *Themeable HTML components*

Description

Themeable HTML components use Sass to generate CSS rules from Bootstrap Sass variables, functions, and/or mixins (i.e., stuff inside of theme). bs_dependencies() makes it a bit easier to create themeable components by compiling sass::sass() (input) together with Bootstrap Sass inside of a theme, and packaging up the result into an htmltools::htmlDependency().

Themable components can also be *dynamically* themed inside of Shiny (i.e., they may be themed in 'real-time' via bs_themer(), and more generally, update their styles in response to shiny::session's setCurrentTheme() method). Dynamically themeable components provide a "recipe" (i.e., a function) to bs_dependency_defer(), describing how to generate new CSS stylesheet(s) from a new theme. This function is called when the HTML page is first rendered, and may be invoked again with a new theme whenever shiny::session's setCurrentTheme() is called.

Usage

```
bs_dependency(
  input = list(),
  theme,
  name,
  version,
  cache_key_extra = NULL,
  .dep_args = list(),
  .sass_args = list()
)

bs_dependency_defer(func, memoise = TRUE)
```

Arguments

input	Sass rules to compile, using theme.
theme	A bs_theme() object.
name	Library name
version	Library version
cache_key_extra	Extra information to add to the sass cache key. It is useful to add the version of your package.
.dep_args	A list of additional arguments to pass to htmltools::htmlDependency(). Note that package has no effect and script must be absolute path(s).
.sass_args	A list of additional arguments to pass to sass::sass_partial().
func	a *non-anonymous* function, with a *single* argument. This function should accept a bs_theme() object and return a single htmltools::htmlDependency(), a list of them, or NULL.
memoise	whether or not to memoise (i.e., cache) func results for a short period of time. The default, TRUE, can have large performance benefits when many instances of the same themable widget are rendered. Note that you may want to avoid memoisation if func relies on side-effects (e.g., files on-disk) that need to change for each themable widget instance.

Value

bs_dependency() returns an htmltools::htmlDependency() and bs_dependency_defer() returns an htmltools::tagFunction()

References

- Theming: Custom components gives a tutorial on creating a dynamically themable custom component.

See Also

Other Bootstrap theme functions: bs_add_variables(), bs_current_theme(), bs_global_theme(), bs_remove(), bs_theme(), bs_theme_dependencies(), bs_theme_preview()

Examples

```
myWidgetVersion <- "1.2.3"

myWidgetDependency <- function() {
  list(
    bs_dependency_defer(myWidgetCss),
    htmlDependency(
      name = "mywidget-js",
      version = myWidgetVersion,
      src = system.file(package = "mypackage", "js"),
      script = "mywidget.js"
    )
  )
}

myWidgetCSS <- function(theme) {
  if (!is_bs_theme(theme)) {
    return(
      htmlDependency(
        name = "mywidget-css",
        version = myWidgetVersion,
        src = system.file(package = "mypackage", "css"),
        stylesheet = "mywidget.css"
      )
    )
  }

  # Compile mywidget.scss using the variables and defaults from the theme
  # object.
  sass_input <- sass::sass_file(system.file(package = "mypackage", "scss/mywidget.scss"))

  bs_dependency(
    input = sass_input,
    theme = theme,
    name = "mywidget",
    version = myWidgetVersion,
    cache_key_extra = utils::packageVersion("mypackage")
  )
}

# Note that myWidgetDependency is not defined inside of myWidget. This is so
# that, if `myWidget()` is called multiple times, Shiny can tell that the
# function objects are identical and deduplicate them.
myWidget <- function(id) {
  div(
    id = id,
    span("myWidget"),
    myWidgetDependency()
  )
}
```

bs_get_variables *Retrieve Sass variable values from the current theme*

Description

Useful for retrieving a variable from the current theme and using the value to inform another R function.

Usage

```
bs_get_variables(theme, varnames)

bs_get_contrast(theme, varnames)
```

Arguments

theme	A bs_theme() object.
varnames	A character string referencing a Sass variable in the current theme.

Value

Returns a character string containing a CSS/Sass value. If the variable(s) are not defined, their value is NA.

References

Theming: Bootstrap 5 variables provides a searchable reference of all theming variables available in Bootstrap 5.

See Also

Other Bootstrap theme utility functions: bootswatch_themes(), builtin_themes(), theme_bootswatch(), theme_version(), versions()

Examples

```
vars <- c("body-bg", "body-color", "primary", "border-radius")
bs_get_variables(bs_theme(), varnames = vars)
bs_get_variables(bs_theme(bootswatch = "darkly"), varnames = vars)

bs_get_contrast(bs_theme(), c("primary", "dark", "light"))

library(htmltools)
div(
  class = "bg-primary",
  style = css(
    color = bs_get_contrast(bs_theme(), "primary")
  )
```

```
)
```

`bs_global_theme` *Global theming*

Description

`bs_global_theme()` creates and sets the global Bootstrap Sass theme. This theme is typically found by `bs_theme_dependencies()` in the app or document where the global theme is being used. You can obtain the current global theme with `bs_global_get()` or directly set the global theme with `bs_global_set()`.

Usage

```
bs_global_theme(
  version = version_default(),
  preset = NULL,
  bg = NULL,
  fg = NULL,
  primary = NULL,
  secondary = NULL,
  success = NULL,
  info = NULL,
  warning = NULL,
  danger = NULL,
  base_font = NULL,
  code_font = NULL,
  heading_font = NULL,
  ...,
  bootswatch = NULL
)

bs_global_set(theme = bs_theme())

bs_global_get()

bs_global_clear()

bs_global_add_variables(
  ...,
  .where = "defaults",
  .default_flag = identical(.where, "defaults")
)

bs_global_add_rules(...)
```

```
bs_global_bundle(...)

bs_global_theme_update(
  ...,
  preset = NULL,
  bg = NULL,
  fg = NULL,
  primary = NULL,
  secondary = NULL,
  success = NULL,
  info = NULL,
  warning = NULL,
  danger = NULL,
  base_font = NULL,
  code_font = NULL,
  heading_font = NULL,
  bootswatch = NULL
)
```

Arguments

version	The major version of Bootstrap to use (see versions() for possible values). Defaults to the currently recommended version for new projects (currently Bootstrap 5).
preset	The name of a theme preset, either a built-in theme provided by bslib or a Bootswatch theme (see builtin_themes() and bootswatch_themes() for possible values). This argument takes precedence over the bootswatch argument and only one preset or bootswatch can be provided. When no bootswatch theme is specified, and version is 5 or higher, preset defaults to "shiny". To remove the "shiny" preset, provide a value of "bootstrap" (this value will also work in bs_theme_update() to remove a preset or bootswatch theme).
bg	A color string for the background.
fg	A color string for the foreground.
primary	A color to be used for hyperlinks, to indicate primary/default actions, and to show active selection state in some Bootstrap components. Generally a bold, saturated color that contrasts with the theme's base colors.
secondary	A color for components and messages that don't need to stand out. (Not supported in Bootstrap 3.)
success	A color for messages that indicate an operation has succeeded. Typically green.
info	A color for messages that are informative but not critical. Typically a shade of blue-green.
warning	A color for warning messages. Typically yellow.
danger	A color for errors. Typically red.
base_font	The default typeface.
code_font	The typeface to be used for code. Be sure this is monospace!

heading_font	The typeface to be used for heading elements.
...	arguments passed along to bs_add_variables().
bootswatch	The name of a bootswatch theme (see bootswatch_themes() for possible values). When provided to bs_theme_update(), any previous Bootswatch theme is first removed before the new one is applied (use bootswatch = "bootstrap" to effectively remove the Bootswatch theme).
theme	A bs_theme() object.
.where	Whether to place the variable definitions before other Sass "defaults", after other Sass "declarations", or after other Sass "rules".
.default_flag	Whether or not to add a !default flag (if missing) to variable expressions. It's recommended to keep this as TRUE when .where = "defaults".

Value

Functions that modify the global theme (e.g., bs_global_set()) invisibly return the previously set theme. bs_global_get() returns the current global theme.

See Also

Other Bootstrap theme functions: bs_add_variables(), bs_current_theme(), bs_dependency(), bs_remove(), bs_theme(), bs_theme_dependencies(), bs_theme_preview()

Examples

```
# Remember the global state now (so we can restore later)
theme <- bs_global_get()

# Use Bootstrap 3 (globally) with some theme customization
bs_global_theme(3, bg = "#444", fg = "#e4e4e4", primary = "#e39777")
if (rlang::is_interactive()) {
  bs_theme_preview(with_themer = FALSE)
}

# If no global theme is active, bs_global_get() returns NULL
bs_global_clear()
bs_global_get()

# Restore the original state
bs_global_set(theme)
```

---

bs_remove *Remove or retrieve Sass code from a theme*

Description

A Bootstrap theme created with bs_theme() is comprised of many Sass layers. bs_remove() and bs_retrieve() allow you to remove or retrieve an individual layer, either to reduce the size of the compiled CSS or to extract styles from a theme.

Usage

```
bs_remove(theme, ids = character(0))

bs_retrieve(theme, ids = character(0), include_unnamed = TRUE)
```

Arguments

theme	A bs_theme() object.
ids	a character vector of ids
include_unnamed	whether or not to include unnamed sass::sass_layer()s (e.g., Bootstrap Sass variables, functions, and mixins).

Value

Returns a modified bs_theme() object.

See Also

Other Bootstrap theme functions: bs_add_variables(), bs_current_theme(), bs_dependency(), bs_global_theme(), bs_theme(), bs_theme_dependencies(), bs_theme_preview()

Examples

```
bs4 <- bs_theme(version = 4)

# Retrieve sass bundle for print styles
bs_retrieve(bs4, "_print", include_unnamed = FALSE)

# Remove CSS rules for print and carousels
bs4_no_print <- bs_remove(bs4, c("_print", "_carousel"))
suppressWarnings(
  bs_retrieve(bs4_no_print, "_print", include_unnamed = FALSE)
)

# Remove BS3 compatibility layer
bs4_no_compat <- bs_remove(bs4, "bs3compat")
```

---

bs_theme *Create a Bootstrap theme*

Description

Creates a Bootstrap theme object, where you can:

- Choose a (major) Bootstrap version.
- Choose a Bootswatch theme (optional).

- Customize main colors and fonts via explicitly named arguments (e.g., `bg`, `fg`, `primary`, etc).
- Customize other, lower-level, Bootstrap Sass variable defaults via `...`.

To learn more about how to implement custom themes, as well as how to use them inside Shiny and R Markdown, see here.

Usage

```
bs_theme(
  version = version_default(),
  preset = NULL,
  ...,
  brand = NULL,
  bg = NULL,
  fg = NULL,
  primary = NULL,
  secondary = NULL,
  success = NULL,
  info = NULL,
  warning = NULL,
  danger = NULL,
  base_font = NULL,
  code_font = NULL,
  heading_font = NULL,
  font_scale = NULL,
  bootswatch = NULL
)

bs_theme_update(
  theme,
  ...,
  preset = NULL,
  bg = NULL,
  fg = NULL,
  primary = NULL,
  secondary = NULL,
  success = NULL,
  info = NULL,
  warning = NULL,
  danger = NULL,
  base_font = NULL,
  code_font = NULL,
  heading_font = NULL,
  font_scale = NULL,
  bootswatch = NULL
)

is_bs_theme(x)
```

Arguments

version	The major version of Bootstrap to use (see versions() for possible values). Defaults to the currently recommended version for new projects (currently Bootstrap 5).
preset	The name of a theme preset, either a built-in theme provided by bslib or a Bootswatch theme (see builtin_themes() and bootswatch_themes() for possible values). This argument takes precedence over the bootswatch argument and only one preset or bootswatch can be provided. When no bootswatch theme is specified, and version is 5 or higher, preset defaults to "shiny". To remove the "shiny" preset, provide a value of "bootstrap" (this value will also work in bs_theme_update() to remove a preset or bootswatch theme).
...	arguments passed along to bs_add_variables().
brand	Specifies how to apply branding to your theme using brand.yml, a simple YAML file that defines key brand elements like colors, fonts, and logos. Valid options: • NULL (default): Automatically looks for a _brand.yml file in the current directory or in _brand/ or brand/ in the current directory. If not found, it searches parent project directories for a _brand.yml file (also possibly in _brand/ or brand/). If a _brand.yml file is found, it is applied to the Bootstrap theme. • TRUE (default): Automatically looks for a _brand.yml file in the current or app directory as described above. If a _brand.yml file *is not found*, bs_theme() will throw an error. • FALSE: Disables any brand.yml usage, even if a _brand.yml file is present. • A file path that directly points to a specific brand.yml file (with any file name) that you want to use. • Use a list to directly provide brand settings directly in R, following the brand.yml structure. Learn more about creating and using brand.yml files at the brand.yml homepage or run shiny::runExample("brand.yml", package = "bslib") to try brand.yml in a demo app.
bg	A color string for the background.
fg	A color string for the foreground.
primary	A color to be used for hyperlinks, to indicate primary/default actions, and to show active selection state in some Bootstrap components. Generally a bold, saturated color that contrasts with the theme's base colors.
secondary	A color for components and messages that don't need to stand out. (Not supported in Bootstrap 3.)
success	A color for messages that indicate an operation has succeeded. Typically green.
info	A color for messages that are informative but not critical. Typically a shade of blue-green.
warning	A color for warning messages. Typically yellow.
danger	A color for errors. Typically red.
base_font	The default typeface.

code_font	The typeface to be used for code. Be sure this is monospace!
heading_font	The typeface to be used for heading elements.
font_scale	A scalar multiplier to apply to the base font size. For example, a value of 1.5 scales font sizes to 150% and a value of 0.8 scales to 80%. Must be a positive number.
bootswatch	The name of a bootswatch theme (see bootswatch_themes() for possible values). When provided to bs_theme_update(), any previous Bootswatch theme is first removed before the new one is applied (use bootswatch = "bootstrap" to effectively remove the Bootswatch theme).
theme	A bs_theme() object.
x	an object.

Value

Returns a sass::sass_bundle() (list-like) object.

Colors

Colors may be provided in any format that htmltools::parseCssColors() can understand. To control the vast majority of the ('grayscale') color defaults, specify both the fg (foreground) and bg (background) colors. The primary and secondary theme colors are also useful for accenting the main grayscale colors in things like hyperlinks, tabset panels, and buttons.

Fonts

Use base_font, code_font, and heading_font to control the main typefaces. These arguments set new defaults for the relevant font-family CSS properties, but don't necessarily import the relevant font files. To both set CSS properties *and* import font files, consider using the various font_face() helpers.

Each *_font argument may be a single font or a font_collection(). A font can be created with font_google(), font_link(), or font_face(), or it can be a character vector of font names in the following format:

- A single unquoted name (e.g., "Source Sans Pro").
- A single quoted name (e.g., "'Source Sans Pro'").
- A comma-separated list of names w/ individual names quoted as necessary. (e.g. c("Open Sans", "'Source Sans Pro'", "'Helvetica Neue', Helvetica, sans-serif"))

font_google() sets local = TRUE by default, which ensures that the font files are downloaded from Google Fonts when your document or app is rendered. This guarantees that the client has access to the font family, making it relatively safe to specify just one font family:

```
bs_theme(base_font = font_google("Pacifico", local = TRUE))
```

That said, we recommend you specify multiple "fallback" font families, especially when relying on remote and/or system fonts being available. Fallback fonts are useful not only for handling missing fonts, but also ensure that your users don't experience a Flash of Invisible Text (FOIT) which can be quite noticeable with remote web fonts on a slow internet connection.

```
bs_theme(base_font = font_collection(font_google("Pacifico", local = FALSE), "Roboto", "sans-serif"))
```

References

- Get Started: Theming introduces theming with bslib in Shiny apps and R Markdown documents.
- Theming: Bootstrap 5 variables provides a searchable reference of all theming variables available in Bootstrap 5.
- Theming: Custom components gives a tutorial on creating a dynamically themable custom component.
- bslib's theming capabilities are powered by the sass package.
- Bootstrap's utility classes can be helpful when you want to change the appearance of an element without writing CSS or customizing your `bs_theme()`.

See Also

Other Bootstrap theme functions: `bs_add_variables()`, `bs_current_theme()`, `bs_dependency()`, `bs_global_theme()`, `bs_remove()`, `bs_theme_dependencies()`, `bs_theme_preview()`

Examples

```
theme <- bs_theme(
  # Controls the default grayscale palette
  bg = "#202123", fg = "#B8BCC2",
  # Controls the accent (e.g., hyperlink, button, etc) colors
  primary = "#EA80FC", secondary = "#48DAC6",
  base_font = c("Grandstander", "sans-serif"),
  code_font = c("Courier", "monospace"),
  heading_font = "'Helvetica Neue', Helvetica, sans-serif",
  # Can also add lower-level customization
  "input-border-color" = "#EA80FC"
)

bs_theme_preview(theme)

# Lower-level bs_add_*() functions allow you to work more
# directly with the underlying Sass code
theme <- bs_add_variables(theme, "my-class-color" = "red")
theme <- bs_add_rules(theme, ".my-class { color: $my-class-color }")
```

`bs_theme_dependencies` *Compile Bootstrap Sass with (optional) theming*

Description

`bs_theme_dependencies()` compiles Bootstrap Sass into CSS and returns it, along with other HTML dependencies, as a list of `htmltools::htmlDependency()`s. Most users won't need to call this function directly as Shiny & R Markdown will perform this compilation automatically when handed a `bs_theme()`. If you're here looking to create a themeable component, see `bs_dependency()`.

Usage

```
bs_theme_dependencies(
  theme,
  sass_options = sass::sass_options_get(output_style = "compressed"),
  cache = sass::sass_cache_get(),
  jquery = jquerylib::jquery_core(3),
  precompiled = get_precompiled_option("bslib.precompiled", default = TRUE)
)
```

Arguments

theme	A bs_theme() object.
sass_options	a sass::sass_options() object.
cache	This can be a directory to use for the cache, a FileCache object created by sass_file_cache(), or FALSE or NULL for no caching.
jquery	a jquerylib::jquery_core() object.
precompiled	Before compiling the theme object, first look for a precompiled CSS file for the theme_version(). If precompiled = TRUE and a precompiled CSS file exists for the theme object, it will be fetched immediately and not compiled. At the moment, we only provide precompiled CSS for "stock" builds of Bootstrap (i.e., no theming additions, Bootswatch themes, or non-default sass_options).

Value

Returns a list of HTML dependencies containing Bootstrap CSS, Bootstrap JavaScript, and jquery. This list may contain additional HTML dependencies if bundled with the theme.

Sass caching and precompilation

If Shiny Developer Mode is enabled (by setting options(shiny.devmode = TRUE) or calling shiny::devmode(TRUE)), both **sass** caching and **bslib** precompilation are disabled by default; that is, a call to bs_theme_dependencies(theme) expands to bs_theme_dependencies(theme, cache = F, precompiled = F)). This is useful for local development as enabling caching/precompilation may produce incorrect results if local changes are made to bslib's source files.

See Also

Other Bootstrap theme functions: bs_add_variables(), bs_current_theme(), bs_dependency(), bs_global_theme(), bs_remove(), bs_theme(), bs_theme_preview()

Examples

```
# Function to preview the styling a (primary) Bootstrap button
library(htmltools)
button <- tags$a(class = "btn btn-primary", href = "#", role = "button", "Hello")
preview_button <- function(theme) {
  browsable(tags$body(bs_theme_dependencies(theme), button))
```

```
}

# Latest Bootstrap
preview_button(bs_theme())
# Bootstrap 3
preview_button(bs_theme(3))
# Bootswatch 4 minty theme
preview_button(bs_theme(4, bootswatch = "minty"))
# Bootswatch 4 sketchy theme
preview_button(bs_theme(4, bootswatch = "sketchy"))
```

`bs_theme_preview` *Preview a Bootstrap theme*

Description

Launches an example shiny app that can be used to get a quick preview of a `bs_theme()`, as well as an interactive GUI for tweaking some of the main theme settings. Calling `bs_theme_preview()` with no arguments starts the theme preview app with the default theme, which is a great way to see the available theme presets or to start creating your own theme.

Usage

```
bs_theme_preview(theme = bs_theme(), ..., with_themer = TRUE)
```

Arguments

`theme`	A `bs_theme()` object.
`...`	passed along to `shiny::runApp()`.
`with_themer`	whether or not to run the app with `run_with_themer()`.

Details

The app that this launches is subject to change as new features are developed in **bslib** and **shiny**.

Value

nothing, this function is called for its side-effects (launching an application).

See Also

Use `run_with_themer()` or `bs_themer()` to add the theming UI to an existing shiny app.

Other Bootstrap theme functions: `bs_add_variables()`, `bs_current_theme()`, `bs_dependency()`, `bs_global_theme()`, `bs_remove()`, `bs_theme()`, `bs_theme_dependencies()`

Examples

```
theme <- bs_theme(bg = "#6c757d", fg = "white", primary = "orange")
bs_theme_preview(theme)
```

`builtin_themes` *Obtain a list of all available built-in* **bslib** *themes.*

Description

Obtain a list of all available built-in **bslib** themes.

Usage

```
builtin_themes(version = version_default(), full_path = FALSE)
```

Arguments

`version`	the major version of Bootstrap.
`full_path`	whether to return a path to the installed theme.

Value

Returns a character vector of built-in themes provided by **bslib**.

See Also

Other Bootstrap theme utility functions: `bootswatch_themes()`, `bs_get_variables()`, `theme_bootswatch()`, `theme_version()`, `versions()`

`card` *A Bootstrap card component*

Description

A general purpose container for grouping related UI elements together with a border and optional padding. To learn more about `card()`s, see the Cards article or the other articles listed in the *References* section below.

Usage

```
card(
  ...,
  full_screen = FALSE,
  height = NULL,
  max_height = NULL,
  min_height = NULL,
  fill = TRUE,
  class = NULL,
  wrapper = card_body,
  id = NULL
)
```

Arguments

`...`	Unnamed arguments can be any valid child of an htmltools tag (which includes card items such as `card_body()`. Named arguments become HTML attributes on returned UI element.
`full_screen`	If `TRUE`, an icon will appear when hovering over the card body. Clicking the icon expands the card to fit viewport size.
`height`	Any valid CSS unit (e.g., `height="200px"`). Doesn't apply when a card is made `full_screen` (in this case, consider setting a `height` in `card_body()`).
`max_height, min_height`	Any valid CSS unit (e.g., `max_height="200px"`). Doesn't apply when a card is made `full_screen` (in this case, consider setting a `max_height` in `card_body()`).
`fill`	Whether or not to allow the card to grow/shrink to fit a fillable container with an opinionated height (e.g., `page_fillable()`).
`class`	Additional CSS classes for the returned UI element.
`wrapper`	A function (which returns a UI element) to call on unnamed arguments in `...` which are not already card item(s) (like `card_header()`, `card_body()`, etc.). Note that non-card items are grouped together into one `wrapper` call (e.g. given `card("a", "b", card_body("c"), "d")`, `wrapper` would be called twice, once with `"a"` and `"b"` and once with `"d"`).
`id`	Provide a unique identifier for the `card()` or `value_box()` to report its full screen state to Shiny. For example, using `id = "my_card"`, you can observe the card's full screen state with `input$my_card_full_screen`.

Value

A `htmltools::div()` tag.

References

Several articles on the bslib website feature the card component:

- Cards
- Get Started: Dashboards

- Get Started: Any Project
- Column-based layouts
- Filling layouts: Full-screen cards

See Also

Card item functions create the various parts of a card.

`navset_card_tab()`, `navset_card_pill()` and `navset_card_underline()` create cards with tabbed navigation.

`layout_columns()` and `layout_column_wrap()` help position multiple cards into columns and rows and can also be used inside a card.

`layout_sidebar()` adds a sidebar to a card when nested in `card()` or `card_body()`.

`value_box()` uses `card()` to highlight a showcase a key piece of information.

Other Components: `accordion()`, `popover()`, `tooltip()`, `value_box()`

Examples

```
library(htmltools)

card(
  full_screen = TRUE,
  card_header(
    "This is the header"
  ),
  card_body(
    p("This is the body."),
    p("This is still the body.")
  ),
  card_footer(
    "This is the footer"
  )
)
```

---

card_body *Card items*

Description

Components designed to be provided as direct children of a `card()`. For a general overview of the `card()` API, see the Cards article or the other articles listed in the *References* section of the `card()` documentation.

Usage

```
card_body(
  ...,
  fillable = TRUE,
  min_height = NULL,
  max_height = NULL,
  max_height_full_screen = max_height,
  height = NULL,
  padding = NULL,
  gap = NULL,
  fill = TRUE,
  class = NULL
)

card_title(..., container = htmltools::h5)

card_header(..., gap = NULL, class = NULL, container = htmltools::div)

card_footer(..., class = NULL)

card_image(
  file,
  ...,
  alt = "",
  src = NULL,
  href = NULL,
  border_radius = c("auto", "top", "bottom", "all", "none"),
  mime_type = NULL,
  class = NULL,
  height = NULL,
  fill = FALSE,
  width = NULL,
  container = NULL
)

as.card_item(x)

is.card_item(x)
```

Arguments

`...`	Unnamed arguments can be any valid child of an htmltools tag. Named arguments become HTML attributes on returned UI element.
`fillable`	Whether or not the card item should be a fillable (i.e. flexbox) container.
`min_height`, `max_height`, `max_height_full_screen`	Any valid CSS length unit.
`height`	Any valid CSS unit (e.g., `height="200px"`). Doesn't apply when a card is made `full_screen` (in this case, consider setting a `height` in `card_body()`).

padding — Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.

gap — A CSS length unit defining the gap (i.e., spacing) between elements provided to .... This argument is only applicable when fillable = TRUE

fill — Whether to allow this element to grow/shrink to fit its card() container.

class — Additional CSS classes for the returned UI element.

container — A function to generate an HTML element to contain the image. Setting this value to card_body() places the image inside the card body area, otherwise the image will extend to the edges of the card.

file — A file path pointing an image. Local images (i.e. not a URI starting with https:// or similar) will be base64 encoded and provided to the src attribute of the <img>. Alternatively, you may directly set the image src, in which case file is ignored.

alt — Alternate text for the image, used by screen readers and assistive devices. Provide alt text with a description of the image for any images with important content. If alt text is not provided, the image will be considered to be decorative and will not be read or announced by screen readers.

For more information, the Web Accessibility Initiative (WAI) has a helpful tutorial on alt text.

src — The src attribute of the <img> tag. If provided, file is ignored entirely. Use src to provide a relative path to a file that will be served by the Shiny application and should not be base64 encoded.

href — An optional URL to link to when a user clicks on the image.

border_radius — Which side of the image should have rounded corners, useful when card_image() is used as an image cap at the top or bottom of the card.

The value of border_radius determines whether the card-img-top ("top"), card-img-bottom ("bottom"), or card-img ("all") Bootstrap classes are applied to the card. The default "auto" value will use the image's position within a card() to automatically choose the appropriate class.

mime_type — The mime type of the file when it is base64 encoded. This argument is available for advanced use cases where mime::guess_type() is unable to automatically determine the file type.

width — Any valid CSS unit (e.g., width="100%").

x — an object to test (or coerce to) a card item.

Value

An htmltools::div() tag.

Functions

- `card_body()`: A general container for the "main content" of a `card()`.
- `card_title()`: Similar to `card_header()` but without the border and background color.
- `card_header()`: A header (with border and background color) for the `card()`. Typically appears before a `card_body()`.
- `card_footer()`: A header (with border and background color) for the `card()`. Typically appears after a `card_body()`.
- `card_image()`: Include static images in a card, for example as an image cap at the top or bottom of the card.
- `as.card_item()`: Mark an object as a card item. This will prevent the `card()` from putting the object inside a `wrapper` (i.e., a `card_body()`).

See Also

`card()` creates a card component.

`navset_card_tab()`, `navset_card_pill()` and `navset_card_underline()` create cards with tabbed navigation.

`layout_columns()` and `layout_column_wrap()` help position multiple cards into columns and rows and can also be used inside a card.

`layout_sidebar()` adds a sidebar to a card when nested in `card()` or `card_body()`.

---

font_face *Helpers for importing web fonts*

Description

`font_google()`, `font_link()`, and `font_face()` are all re-exported from the **sass** package (see `sass::font_face()` for details). For a quick example of how to use these functions with `bs_theme()`, see the examples section below.

Examples

```
# If you have an internet connection, running the following code
# will download, cache, and import the relevant Google Font files
# for local use
theme <- bs_theme(
  base_font = font_google("Fira Sans"),
  code_font = font_google("Fira Code"),
  heading_font = font_google("Fredoka One")
)
if (interactive()) {
  bs_theme_preview(theme)
}

# Three different yet equivalent ways of importing a remotely-hosted Google Font
```

```
a <- font_google("Crimson Pro", wght = "200..900", local = FALSE)
b <- font_link(
  "Crimson Pro",
  href = "https://fonts.googleapis.com/css2?family=Crimson+Pro:wght@200..900"
)
url <- "https://fonts.gstatic.com/s/crimsonpro/v13/q5uDsoa5M_tv7IihmnkabARboYF6CsKj.woff2"
c <- font_face(
  family = "Crimson Pro",
  style = "normal",
  weight = "200 900",
  src = paste0("url(", url, ") format('woff2')")
)
theme <- bs_theme(base_font = c)
if (interactive()) {
  bs_theme_preview(theme)
}
```

`input_code_editor` *Code editor input*

Description

Creates an interactive light-weight code editor input that can be used in Shiny applications. The editor provides syntax highlighting, line numbers, and other basic code editing features powered by Prism Code Editor. For a complete example, run `shiny::runExample("code-editor", package = "bslib")`.

The editor value is not sent to R on every keystroke. Instead, updates are reflected on the server when the user moves away from the editor or when they press `Ctrl/Cmd + Enter`.

Note that this input is not designed for editing or rendering large files. Displaying files with 1,000 lines or more may lead to performance issues.

Usage

```
input_code_editor(
  id,
  label = NULL,
  value = "",
  ...,
  language = "plain",
  height = "auto",
  width = "100%",
  theme_light = "github-light",
  theme_dark = "github-dark",
  read_only = FALSE,
  line_numbers = NULL,
  word_wrap = NULL,
  tab_size = 2,
  indentation = c("space", "tab"),
```

```
  fill = TRUE
)

update_code_editor(
  id,
  ...,
  value = NULL,
  label = NULL,
  language = NULL,
  theme_light = NULL,
  theme_dark = NULL,
  read_only = NULL,
  line_numbers = NULL,
  word_wrap = NULL,
  tab_size = NULL,
  indentation = NULL,
  session = get_current_session()
)
```

Arguments

id	Input ID. Access the current value with input$<id>.
label	Display label for the input. Default is NULL for no label.
value	Code content. Default is an empty string.
...	Named arguments, e.g. class and style, that will be added to the outer container of the input.
language	Programming language for syntax highlighting. Supported languages include "r", "python", "julia", "sql", "javascript", "typescript", "html", "css", "scss", "sass", "json", "markdown", "yaml", "xml", "toml", "ini", "bash", "docker", "latex", "cpp", "rust", "diff", and "plain". Default is "plain".
height	CSS height of the editor. Default is "300px".
width	CSS width of the editor. Default is "100%".
theme_light, theme_dark	Theme to use in light or dark mode. Defaults to "github-light" and "github-dark", respectively. See the Theme section for more details.
read_only	Whether the editor should be read-only. Default is FALSE.
line_numbers	Whether to show line numbers. Default is TRUE, except for markdown and plain text.
word_wrap	Whether to wrap long lines, by default disabled when line_numbers is FALSE.
tab_size	Number of spaces per tab. Default is 2.
indentation	Type of indentation: "space" or "tab". Default is "space".
fill	Whether or not to allow the card to grow/shrink to fit a fillable container with an opinionated height (e.g., page_fillable()).
session	a shiny session object (the default should almost always be used).

Value

An HTML tag object that can be included in a Shiny UI.

Keyboard shortcuts

The editor supports the following keyboard shortcuts:

- `Ctrl/Cmd+Enter`: Submit the current code to R
- `Ctrl/Cmd+Z`: Undo
- `Ctrl/Cmd+Shift+Z`: Redo
- `Tab`: Indent selection
- `Shift+Tab`: Dedent selection

Themes

The editor automatically switches between `theme_light` and `theme_dark` when used with `input_dark_mode()`. Otherwise, the editor will use `theme_light` by default.

A variety of themes are available for use. The full list of bundled themes is: `"atom-one-dark"`, `"dracula"`, `"github-dark-dimmed"`, `"github-dark"`, `"github-light"`, `"night-owl-light"`, `"night-owl"`, `"prism-okaidia"`, `"prism-solarized-light"`, `"prism-tomorrow"`, `"prism-twilight"`, `"prism"`, `"vs-code-dark"`, `"vs-code-light"`.

See Also

Other input controls: `input_dark_mode()`, `input_switch()`

Examples

```
library(shiny)
library(bslib)

ui <- page_fluid(
  input_code_editor(
    "sql_query",
    value = "SELECT * FROM table",
    language = "sql"
  ),
  verbatimTextOutput("code_output")
)

server <- function(input, output, session) {
  output$code_output <- renderPrint({
    input$sql_query
  })
}

shinyApp(ui, server)
```

input_dark_mode *Dark mode input control*

Description

Creates a button that toggles between dark and light modes, specifically for toggling between light and dark Bootstrap color modes – a new feature added in Bootstrap 5.3.

Usage

```
input_dark_mode(..., id = NULL, mode = NULL)

toggle_dark_mode(mode = NULL, ..., session = get_current_session())
```

Arguments

`...`	Additional attributes to be passed to the input control UI, such as `class`, `style`, etc. In `toggle_dark_mode()`, the `...` are included for future extensibility and are currently ignored.
`id`	An optional input id, required to reactively read the current color mode.
`mode`	The initial mode of the dark mode switch. By default or when set to `NULL`, the user's system settings for preferred color scheme will be used. Otherwise, set to `"light"` or `"dark"` to force a particular initial mode.
`session`	A Shiny session object (the default should almost always be used).

Value

Returns a UI element for a dark mode switch input control. The server value received for the input corresponding to `id` will be a string value with the current color mode (`"light"` or `"dark"`).

Functions

- `input_dark_mode()`: Create a dark mode switch input control
- `toggle_dark_mode()`: Programmatically toggle or set the current light or dark color mode.

See Also

Other input controls: `input_code_editor()`, `input_switch()`

input_submit_textarea *Create a textarea input control with explicit submission*

Description

Creates a textarea input where users can enter multi-line text and submit their input using a dedicated button or keyboard shortcut. This control is ideal when you want to capture finalized input, rather than reacting to every keystroke, making it useful for chat boxes, comments, or other scenarios where users may compose and review their text before submitting.

Usage

```
input_submit_textarea(
  id,
  label = NULL,
  ...,
  placeholder = NULL,
  value = "",
  width = "min(680px, 100%)",
  rows = 1,
  button = NULL,
  toolbar = NULL,
  submit_key = c("enter+modifier", "enter")
)

update_submit_textarea(
  id,
  ...,
  value = NULL,
  placeholder = NULL,
  label = NULL,
  submit = FALSE,
  focus = FALSE,
  session = get_current_session()
)
```

Arguments

id	The input ID.
label	The label to display above the input control. If NULL, no label is displayed.
...	Additional attributes to apply to the underlying <textarea> element (e.g., spellcheck, autocomplete, etc).
placeholder	The placeholder text for the user input.
value	The value to set the user input to.
width	Any valid CSS unit (e.g., width="100%").

rows	The number of rows (i.e., height) of the textarea. This essentially sets the minimum height – the textarea can grow taller as the user enters more text.
button	A htmltools::tags element to use for the submit button. It's recommended that this be a input_task_button() since it will automatically provide a busy indicator (and disable) until the next flush occurs. Note also that if the submit button launches a shiny::ExtendedTask, this button can also be bound to the task (bind_task_button()) and/or manually updated for more accurate progress reporting (update_task_button()).
toolbar	A list of optional UI elements (e.g., links, icons) to display next to the submit button.
submit_key	A character string indicating what keyboard event should trigger the submit button. The default is enter+modifier, which requires the user to hold down Ctrl (or Cmd on Mac) before pressing Enter to submit. This helps prevent accidental submissions. To allow submission with just the Enter key, use enter. In this case, the user can still insert new lines using Shift+Enter or Alt+Enter.
submit	Whether to automatically submit the text for the user. Requires value.
focus	Whether to move focus to the input element. Requires value.
session	The session object; using the default is recommended.

Value

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

Server value

The server receives a character string containing the user's text input.

Important: The initial server value is always "" (empty string), regardless of any value parameter provided to input_submit_textarea(). The server value updates only when the user explicitly submits the input by either pressing the Enter key (possibly with a modifier key) or clicking the submit button.

See Also

update_submit_textarea(), input_task_button()

Examples

```
library(shiny)
library(bslib)

ui <- page_fluid(
  input_submit_textarea("text", placeholder = "Enter some input..."),
  verbatimTextOutput("value")
)
server <- function(input, output) {
  output$value <- renderText({
    req(input$text)
    Sys.sleep(2)
```

```
    paste("You entered:", input$text)
  })
}
shinyApp(ui, server)
```

`input_switch` *Switch input control*

Description

Create an on-off style switch control for specifying logical values.

Usage

```
input_switch(id, label, value = FALSE, width = NULL)

update_switch(id, label = NULL, value = NULL, session = get_current_session())

toggle_switch(id, value = NULL, session = get_current_session())
```

Arguments

`id`	An input id.
`label`	A label for the switch.
`value`	Whether or not the switch should be checked by default.
`width`	Any valid CSS unit (e.g., `width="200px"`).
`session`	a shiny session object (the default should almost always be used).

Value

Returns a UI element for a switch input control. The server value received for the input corresponding to `id` will be a logical (`TRUE`/`FALSE`) value.

See Also

Other input controls: `input_code_editor()`, `input_dark_mode()`

Examples

```
library(shiny)
library(bslib)

ui <- page_fixed(
  title = "Keyboard Settings",
  h2("Keyboard Settings"),
  input_switch("auto_capitalization", "Auto-Capitalization", TRUE),
  input_switch("auto_correction", "Auto-Correction", TRUE),
```

```
  input_switch("check_spelling", "Check Spelling", TRUE),
  input_switch("smart_punctuation", "Smart Punctuation"),
  h2("Preview"),
  verbatimTextOutput("preview")
)

server <- function(input, output, session) {
  output$preview <- renderPrint({
    list(
      auto_capitalization = input$auto_capitalization,
      auto_correction = input$auto_correction,
      check_spelling = input$check_spelling,
      smart_punctuation = input$smart_punctuation
    )
  })
}

shinyApp(ui, server)
```

`input_task_button` *Button for launching longer-running operations*

Description

`input_task_button` is a button that can be used in conjuction with `shiny::bindEvent()` (or the older `shiny::eventReactive()` and `shiny::observeEvent()` functions) to trigger actions or recomputation.

It is similar to `shiny::actionButton()`, except it prevents the user from clicking when its operation is already in progress.

Upon click, it automatically displays a customizable progress message and disables itself; and after the server has dealt with whatever reactivity is triggered from the click, the button automatically reverts to its original appearance and re-enables itself.

Usage

```
input_task_button(
  id,
  label,
  ...,
  icon = NULL,
  label_busy = "Processing...",
  icon_busy = rlang::missing_arg(),
  type = "primary",
  auto_reset = TRUE
)

update_task_button(id, ..., state = NULL, session = get_current_session())
```

Arguments

id	The input slot that will be used to access the value.
label	The label of the button while it is in ready (clickable) state; usually a string.
...	In input_task_button(), named arguments become attributes to include on the <button> element, e.g. class or data attributes. Unnamed arguments can provide additional states for the button, see the "Custom states" section. In update_task_button(), ... are ignored and must be empty. The task button only supports changing between pre-defined states.
icon	An optional icon to display next to the label while the button is in ready state. See fontawesome::fa_i().
label_busy	The label of the button while it is busy.
icon_busy	The icon to display while the button is busy. By default, fontawesome::fa_i("refresh", class = "fa-spin", "aria-hidden" = "true") is used, which displays a spinning "chasing arrows" icon. You can create spinning icons out of other Font Awesome icons by using the same expression, but replacing "refresh" with a different icon name. See fontawesome::fa_i().
type	One of the Bootstrap theme colors ("primary", "default", "secondary", "success", "danger", "warning", "info", "light", "dark"), or NULL to leave off the Bootstrap-specific button CSS classes altogether.
auto_reset	If TRUE (the default), automatically put the button back in "ready" state after its click is handled by the server.
state	If "busy", put the button into busy/disabled state. If "ready", put the button into ready/enabled state.
session	The session object; using the default is recommended.

Manual button reset

In some advanced use cases, it may be necessary to keep a task button in its busy state even after the normal reactive processing has completed. Calling update_task_button(id, state = "busy") from the server will opt out of any currently pending reset for a specific task button. After doing so, the button can be re-enabled by calling update_task_button(id, state = "ready") after each click's work is complete.

You can also pass an explicit auto_reset = FALSE to input_task_button(), which means that button will *never* be automatically re-enabled and will require update_task_button(id, state = "ready") to be called each time.

Note that, as a general rule, Shiny's update family of functions do not take effect at the instant that they are called, but are held until the end of the current reactive cycle. So if you have many different reactive calculations and outputs, you don't have to be too careful about when you call update_task_button(id, state = "ready"), as the button on the client will not actually re-enable until the same moment that all of the updated outputs simultaneously sent to the client.

Custom states

The task button is designed to automatically switch between two states: the "ready" state, where the button is clickable and displays the label and icon; and the "busy" state, where the button is disabled and displays label_busy and icon_busy.

In advanced use cases, you can include additional states by adding an `htmltools::div()` with a `slot` attribute naming the state and the icon and label as the first and second children, respectively.

```
input_task_button(
  label = "Ring home",
  icon = fontawesome::fa_i("phone"),
  div(slot = "ringing", fontawesome::fa_i("bell"), "Ringing..."),
  div(
    slot = "voice-mail",
    fontawesome::fa_i("voicemail"),
    "Leaving a message..."
  )
)
```

You can move between these states by calling `update_task_button()` and passing the slot name to the `state` argument, e.g. `state="ringing"`. See the section above on manual button resetting, which you will likely need to use in conjunction with custom states.

Server value

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

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

See Also

`bind_task_button()`

Examples

```
library(shiny)
library(bslib)

ui <- page_sidebar(
  sidebar = sidebar(
    open = "always",
    input_task_button("resample", "Resample"),
  ),
  verbatimTextOutput("summary")
)

server <- function(input, output, session) {
  sample <- eventReactive(input$resample, ignoreNULL=FALSE, {
    Sys.sleep(2)  # Make this artificially slow
    rnorm(100)
  })

  output$summary <- renderPrint({
```

```
    summary(sample())
  })
}

shinyApp(ui, server)
```

`layout_columns` *Responsive 12-column grid layouts*

Description

Create responsive, column-based grid layouts, based on a 12-column grid.

Usage

```
layout_columns(
  ...,
  col_widths = NA,
  row_heights = NULL,
  fill = TRUE,
  fillable = TRUE,
  gap = NULL,
  class = NULL,
  height = NULL,
  min_height = NULL,
  max_height = NULL
)
```

Arguments

`...`	Unnamed arguments should be UI elements (e.g., `card()`). Named arguments become attributes on the containing htmltools::tag element.
`col_widths`	One of the following: • `NA` (the default): Automatically determines a sensible number of columns based on the number of children. • A numeric vector of integers between 1 and 12, where each element represents the number of columns for the relevant UI element. Elements that happen to go beyond 12 columns wrap onto the next row. For example, `col_widths = c(4, 8, 12)` allocates 4 columns to the first element, 8 columns to the second element, and 12 columns to the third element (which wraps to the next row). Negative values are also allowed, and are treated as empty columns. For example, `col_widths = c(-2, 8, -2)` would allocate 8 columns to an element (with 2 empty columns on either side). • A `breakpoints()` object, where each breakpoint may be either of the above.
`row_heights`	One of the following:

- A numeric vector, where each value represents the fractional unit (`fr`) height of the relevant row. If there are more rows than values provided, the pattern will repeat. For example, `row_heights = c(1, 2)` allows even rows to take up twice as much space as odd rows.
- A list of numeric and CSS length units, where each value represents the height of the relevant row. If more rows are needed than values provided, the pattern will repeat. For example, `row_heights = list("auto", 1)` allows the height of odd rows to be driven by its contents and even rows to be `1fr`.
- A character vector/string of CSS length units. In this case, the value is supplied directly to `grid-auto-rows`.
- A `breakpoints()` object, where each breakpoint may be either of the above.

`fill` Whether or not to allow the layout to grow/shrink to fit a fillable container with an opinionated height (e.g., `page_fillable()`).

`fillable` Whether or not each element is wrapped in a fillable container.

`gap` A CSS length unit defining the gap (i.e., spacing) between elements provided to `...`. This argument is only applicable when `fillable = TRUE`

`class` Additional CSS classes for the returned UI element.

`height` Any valid CSS unit (e.g., `height="200px"`). Doesn't apply when a card is made `full_screen` (in this case, consider setting a `height` in `card_body()`).

`min_height, max_height`
The maximum or minimum height of the layout container. Can be any valid CSS unit (e.g., `max_height="200px"`). Use these arguments in filling layouts to ensure that a layout container doesn't shrink below `min_height` or grow beyond `max_height`.

References

Column-based layouts on the bslib website.

See Also

`breakpoints()` for more information on specifying column widths at responsive breakpoints.

Other Column layouts: `layout_column_wrap()`

Examples

```
x <- card("A simple card")

page_fillable(
  layout_columns(x, x, x, x)
)

# Or add a list of items, spliced with rlang's `!!!` operator
page_fillable(
 layout_columns(!!!list(x, x, x))
)
```

```
page_fillable(
  layout_columns(
    col_widths = c(6, 6, 12),
    x, x, x
  )
)

page_fillable(
  layout_columns(
    col_widths = c(6, 6, -2, 8),
    row_heights = c(1, 3),
    x, x, x
  )
)

page_fillable(
  fillable_mobile = TRUE,
  layout_columns(
    col_widths = breakpoints(
      sm = c(12, 12, 12),
      md = c(6, 6, 12),
      lg = c(4, 4, 4)
    ),
    x, x, x
  )
)
```

`layout_column_wrap` *Column-first uniform grid layouts*

Description

Wraps a 1d sequence of UI elements into a 2d grid. The number of columns (and rows) in the grid dependent on the column `width` as well as the size of the display. For more explanation and illustrative examples, see the *References* section below.

Usage

```
layout_column_wrap(
  ...,
  width = "200px",
  fixed_width = FALSE,
  heights_equal = c("all", "row"),
  fill = TRUE,
  fillable = TRUE,
  height = NULL,
  height_mobile = NULL,
  min_height = NULL,
```

```
  max_height = NULL,
  gap = NULL,
  class = NULL
)
```

Arguments

`...`	Unnamed arguments should be UI elements (e.g., `card()`). Named arguments become attributes on the containing htmltools::tag element.
`width`	The desired width of each card, which can be any of the following: • A (unit-less) number between 0 and 1. – This should be specified as `1/num`, where `num` represents the number of desired columns. • A CSS length unit – Either the minimum (when `fixed_width=FALSE`) or fixed width (`fixed_width=TRUE`). • `NULL` – Allows power users to set the `grid-template-columns` CSS property manually, either via a `style` attribute or a CSS stylesheet.
`fixed_width`	When `width` is greater than 1 or is a CSS length unit, e.g. `"200px"`, `fixed_width` indicates whether that `width` value represents the absolute size of each column (`fixed_width=TRUE`) or the minimum size of a column (`fixed_width=FALSE`). When `fixed_width=FALSE`, new columns are added to a row when `width` space is available and columns will never exceed the container or viewport size. When `fixed_width=TRUE`, all columns will be exactly `width` wide, which may result in columns overflowing the parent container.
`heights_equal`	If `"all"` (the default), every card in every row of the grid will have the same height. If `"row"`, then every card in *each* row of the grid will have the same height, but heights may vary between rows.
`fill`	Whether or not to allow the layout to grow/shrink to fit a fillable container with an opinionated height (e.g., `page_fillable()`).
`fillable`	Whether or not each element is wrapped in a fillable container.
`height`	Any valid CSS unit (e.g., `height="200px"`). Doesn't apply when a card is made `full_screen` (in this case, consider setting a `height` in `card_body()`).
`height_mobile`	Any valid CSS unit to use for the height when on mobile devices (or narrow windows).
`min_height, max_height`	The maximum or minimum height of the layout container. Can be any valid CSS unit (e.g., `max_height="200px"`). Use these arguments in filling layouts to ensure that a layout container doesn't shrink below `min_height` or grow beyond `max_height`.
`gap`	A CSS length unit defining the gap (i.e., spacing) between elements provided to `...`. This argument is only applicable when `fillable = TRUE`
`class`	Additional CSS classes for the returned UI element.

References

The bslib website features `layout_column_wrap()` in two places:

- Column-based layouts
- Cards: Multiple cards

See Also

Other Column layouts: `layout_columns()`

Examples

```
x <- card("A simple card")

# Always has 2 columns (on non-mobile)
layout_column_wrap(width = 1/2, x, x, x)

# Automatically lays out three cards into columns
# such that each column is at least 200px wide:
layout_column_wrap(x, x, x)

# To use larger column widths by default, set `width`.
# This example has 3 columns when the screen is at least 900px wide:
layout_column_wrap(width = "300px", x, x, x)

# You can add a list of items, spliced with rlang's `!!!` operator
layout_column_wrap(!!!list(x, x, x))
```

---

nav-items *Navigation items*

Description

Create nav item(s) for use inside nav containers (e.g., `navset_tab()`, `navset_bar()`, etc).

Usage

```
nav_panel(title, ..., value = title, icon = NULL)

nav_panel_hidden(value, ..., icon = NULL)

nav_menu(title, ..., value = title, icon = NULL, align = c("left", "right"))

nav_item(...)

nav_spacer()
```

Arguments

`title`	A title to display. Can be a character string or UI elements (i.e., htmltools::tags).
`...`	Depends on the function: • For `nav_panel()` and `nav_panel_hidden()`: UI elements (i.e., htmltools::tags) to display when the item is active. • For `nav_menu()`: a collection of nav items (e.g., `nav_panel()`, `nav_item()`). • For `nav_item()`: UI elements (i.e., htmltools::tags) to place directly in the navigation panel (e.g., search forms, links to external content, etc).
`value`	A character string to assign to the nav item. This value may be supplied to the relevant container's `selected` argument in order to show particular nav item's content immediately on page load. This value is also useful for programmatically updating the selected content via `nav_select()`, `nav_hide()`, etc (updating selected tabs this way is often useful for showing/hiding panels of content via other UI controls like `shiny::radioButtons()` – in this scenario, consider using `nav_panel_hidden()` with `navset_hidden()`).
`icon`	Optional icon to appear next to the nav item's `title`.
`align`	horizontal alignment of the dropdown menu relative to dropdown toggle.

Value

A nav item that may be passed to a nav container (e.g. `navset_tab()`).

Functions

- `nav_panel()`: Content to display when the given item is selected.
- `nav_panel_hidden()`: Create nav content for use inside `navset_hidden()` (for creating custom navigation controls via `navs_select()`),
- `nav_menu()`: Create a menu of nav items.
- `nav_item()`: Place arbitrary content in the navigation panel (e.g., search forms, links to external content, etc.)
- `nav_spacer()`: Adding spacing between nav items.

See Also

navset create the navigation container holding the nav panels.

`nav_menu()`, `nav_item()`, `nav_spacer()` create menus, items, or space in the navset control area.

`nav_insert()`, `nav_remove()` programmatically add or remove nav panels.

`nav_select()`, `nav_show()`, `nav_hide()` change the state of a `nav_panel()` in a navset.

Other Panel container functions: `nav_select()`, `navset`

navbar_options *Create a set of navbar options*

Description

A navbar_options() object captures options specific to the appearance and behavior of the navbar, independent from the content displayed on the page. This helper should be used to create the list of options expected by navbar_options in page_navbar() and navset_bar().

Usage

```
navbar_options(
  ...,
  position = c("static-top", "fixed-top", "fixed-bottom"),
  bg = NULL,
  theme = c("auto", "light", "dark"),
  collapsible = TRUE,
  underline = TRUE
)
```

Arguments

...	Additional attributes that will be passed directly to the navbar container element.
position	Determines whether the navbar should be displayed at the top of the page with normal scrolling behavior ("static-top"), pinned at the top ("fixed-top"), or pinned at the bottom ("fixed-bottom"). Note that using "fixed-top" or "fixed-bottom" will cause the navbar to overlay your body content, unless you add padding, e.g.: tags$style(type="text/css", "body {padding-top: 70px;}")
bg	a CSS color to use for the navbar's background color.
theme	Either "dark" for a light text color (on a **dark** background) or "light" for a dark text color (on a **light** background). If "auto" (the default) and bg is provided, the best contrast to bg is chosen.
collapsible	TRUE to automatically collapse the navigation elements into an expandable menu on mobile devices or narrow window widths.
underline	Whether or not to add underline styling to page or navbar links when active or focused.

Details

Navbar style with Bootstrap 5 and Bootswatch themes:

In **bslib** v0.9.0, the default navbar colors for Bootswatch themes with Bootstrap 5 changed. Prior to v0.9.0, bslib pre-selected navbar background colors in light and dark mode; after v0.9.0 the default navbar colors are less opinionated by default and follow light or dark mode (see input_dark_mode()).

You can use `navbar_options()` to adjust the colors of the navbar when using a Bootswatch preset theme with Bootstrap 5. For example, the Bootswatch documentation for the Flatly theme shows 4 navbar variations. Inspecting the source code for the first example reveals the following markup:

```
<nav class="navbar navbar-expand-lg bg-primary" data-bs-theme="dark">
  <!-- all of the navbar html -->
</nav>
```

Note that this navbar uses the `bg-primary` class for a dark navy background. The navbar's white text is controlled by the `data-bs-theme="dark"` attribute, which is used by Bootstrap for light text on a *dark* background. In **bslib**, you can achieve this look with:

```
ui <- page_navbar(
  theme = bs_theme(5, "flatly"),
  navbar_options = navbar_options(class = "bg-primary", theme = "dark")
)
```

This particular combination of `class = "bg-primary"` and `theme = "dark"` works well for most Bootswatch presets.

Another variation from the Flatly documentation features a navar with dark text on a light background:

```
ui <- page_navbar(
  theme = bs_theme(5, "flatly"),
  navbar_options = navbar_options(class = "bg-light", theme = "light")
)
```

The above options set navbar foreground and background colors that are always the same in both light and dark modes. To customize the navbar colors used in light or dark mode, you can use the `$navbar-light-bg` and `$navbar-dark-bg` Sass variables. When provided, bslib will automatically choose to use light or dark text as the foreground color.

```
ui <- page_navbar(
  theme = bs_theme(
    5,
    preset = "flatly",
    navbar_light_bg = "#18BC9C", # flatly's success color (teal)
    navbar_dark_bg = "#2C3E50"   # flatly's primary color (navy)
  )
)
```

Finally, you can also use the `$navbar-bg` Sass variable to set the navbar background color for both light and dark modes:

```
ui <- page_navbar(
  theme = bs_theme(
    5,
    preset = "flatly",
    navbar_bg = "#E74C3C" # flatly's danger color (red)
  )
)
```

Value

Returns a list of navbar options.

Changelog

This function was introduced in **bslib** v0.9.0, replacing the `position`, `bg`, `inverse`, `collapsible` and `underline` arguments of `page_navbar()` and `navset_bar()`. Those arguments are deprecated with a warning and will be removed in a future version of **bslib**. Note that the deprecated `inverse` argument of `page_navbar()` and `navset_bar()` was replaced with the `theme` argument of `navbar_options()`.

Examples

```
navbar_options(position = "static-top", bg = "#2e9f7d", underline = FALSE)
```

---

navset *Navigation containers*

Description

Render a collection of `nav_panel()` items into a container.

Usage

```
navset_tab(..., id = NULL, selected = NULL, header = NULL, footer = NULL)

navset_pill(..., id = NULL, selected = NULL, header = NULL, footer = NULL)

navset_underline(..., id = NULL, selected = NULL, header = NULL, footer = NULL)

navset_pill_list(
  ...,
  id = NULL,
  selected = NULL,
  header = NULL,
  footer = NULL,
  well = TRUE,
  fluid = TRUE,
  widths = c(4, 8)
)

navset_hidden(..., id = NULL, selected = NULL, header = NULL, footer = NULL)

navset_bar(
  ...,
  title = NULL,
```

```
  id = NULL,
  selected = NULL,
  sidebar = NULL,
  fillable = TRUE,
  gap = NULL,
  padding = NULL,
  header = NULL,
  footer = NULL,
  fluid = TRUE,
  navbar_options = NULL,
  position = deprecated(),
  bg = deprecated(),
  inverse = deprecated(),
  collapsible = deprecated()
)

navset_card_tab(
  ...,
  id = NULL,
  selected = NULL,
  title = NULL,
  sidebar = NULL,
  header = NULL,
  footer = NULL,
  height = NULL,
  full_screen = FALSE,
  wrapper = card_body
)

navset_card_pill(
  ...,
  id = NULL,
  selected = NULL,
  title = NULL,
  sidebar = NULL,
  header = NULL,
  footer = NULL,
  height = NULL,
  placement = c("above", "below"),
  full_screen = FALSE,
  wrapper = card_body
)

navset_card_underline(
  ...,
  id = NULL,
  selected = NULL,
  title = NULL,
```

```
  sidebar = NULL,
  header = NULL,
  footer = NULL,
  height = NULL,
  full_screen = FALSE,
  wrapper = card_body
)
```

Arguments

...	a collection of nav_panel() items.
id	a character string used for dynamically updating the container (see nav_select()).
selected	a character string matching the value of a particular nav_panel() item to selected by default.
header	UI element(s) (htmltools::tags) to display *above* the nav content. For card-based navsets, these elements are implicitly wrapped in a card_body(). To control things like padding, fill, etc., wrap the elements in an explicit card_body().
footer	UI element(s) (htmltools::tags) to display *below* the nav content. For card-based navsets, these elements are implicitly wrapped in a card_body(). To control things like padding, fill, etc., wrap the elements in an explicit card_body().
well	TRUE to place a well (gray rounded rectangle) around the navigation list.
fluid	TRUE to use fluid layout; FALSE to use fixed layout.
widths	Column widths of the navigation list and tabset content areas respectively.
title	A (left-aligned) title to place in the card header/footer. If provided, other nav items are automatically right aligned.
sidebar	A sidebar() component to display on every nav_panel() page.
fillable	Whether or not to allow fill items to grow/shrink to fit the browser window. If TRUE, all nav_panel() pages are fillable. A character vector, matching the value of nav_panel()s to be filled, may also be provided. Note that, if a sidebar is provided, fillable makes the main content portion fillable.
gap	A CSS length unit defining the gap (i.e., spacing) between elements provided to ....
padding	Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.
navbar_options	Options to control the appearance and behavior of the navbar. Use navbar_options() to create the list of options.
position	**[Deprecated]** Please use navbar_options = navbar_options(position=) instead.
bg	**[Deprecated]** Please use navbar_options = navbar_options(bg=) instead.

inverse	**[Deprecated]** Please use navbar_options = navbar_options(inverse=) instead.
collapsible	**[Deprecated]** Please use navbar_options = navbar_options(collapsible=) instead.
height	Any valid CSS unit (e.g., height="200px"). Doesn't apply when a card is made full_screen (in this case, consider setting a height in card_body()).
full_screen	If TRUE, an icon will appear when hovering over the card body. Clicking the icon expands the card to fit viewport size.
wrapper	A function (which returns a UI element) to call on unnamed arguments in ... which are not already card item(s) (like card_header(), card_body(), etc.). Note that non-card items are grouped together into one wrapper call (e.g. given card("a", "b", card_body("c"), "d"), wrapper would be called twice, once with "a" and "b" and once with "d").
placement	placement of the nav items relative to the content.

Examples

A basic example:

This first example creates a simple tabbed navigation container with two tabs. The tab name and the content of each tab are specified in the nav_panel() calls and navset_tab() creates the tabbed navigation around these two tabs.

```
library(htmltools)

navset_tab(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content."))
)
```

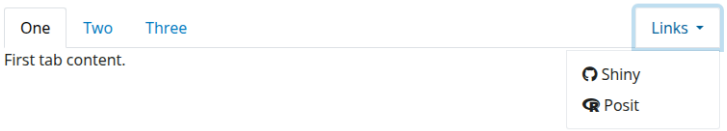
In the rest of the examples, we'll include links among the tabs (or pills) in the navigation controls.

```
link_shiny <- tags$a(shiny::icon("github"), "Shiny", href = "https://github.com/rstudio/shiny", targ
link_posit <- tags$a(shiny::icon("r-project"), "Posit", href = "https://posit.co", target = "_blank")
```

navset_tab():

You can fully customize the controls in the navigation component. In this example, we've added a direct link to the Shiny repository using nav_item(). We've also included a dropdown menu using nav_menu() containing an option to select a third tab panel and another direct link to Posit's website. Finally, we've separated the primary tabs on the left from the direct link and dropdown menu on the right using nav_spacer().

```
navset_tab(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```



`navset_pill()`:

`navset_pill()` creates a navigation container that behaves exactly like `navset_tab()`, but the tab toggles are *pills* or button-shaped.

```
navset_pill(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```

`navset_underline()`:

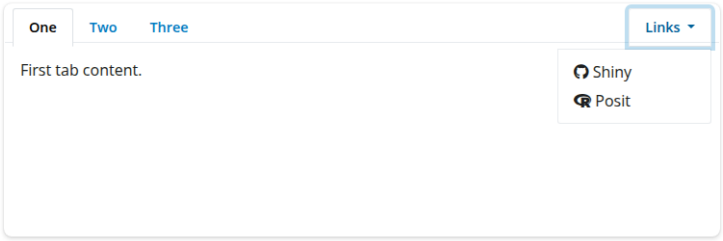
`navset_underline()` creates a navigation container that behaves exactly like `navset_tab()` and `navset_pill()`, but the active/focused navigation links are styled with an underline.

```
navset_underline(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```

`navset_card_tab()`:

The tabbed navigation container can also be used in a `card()` component thanks to `navset_card_tab()`. Learn more about this approach in the article about Cards, including how to add a shared sidebar to all tabs in the card using the `sidebar` argument of `navset_card_tab()`.

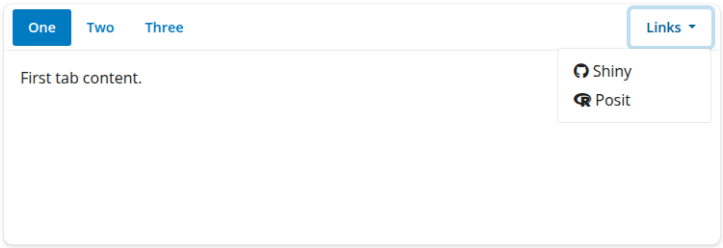
```
navset_card_tab(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```



`navset_card_pill()`:

Similar to `navset_pill()`, `navset_card_pill()` provides a pill-shaped variant to `navset_card_tab()`. You can use the `placement` argument to position the navbar `"above"` or `"below"` the card body.

```
navset_card_pill(
  placement = "above",
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```

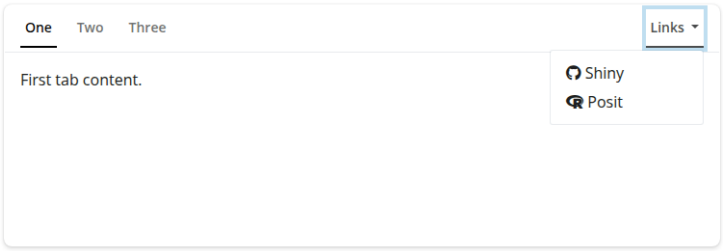


`navset_card_underline()`:

`navset_card_underline()` provides a card-based variant of `navset_underline()`.

```
navset_card_underline(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
```

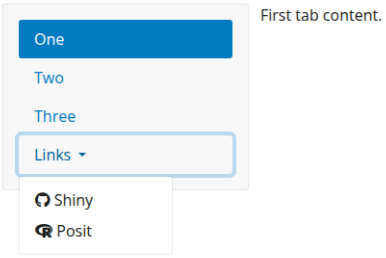
```
  )
)
```



`navset_pill_list()`:

Furthermore, `navset_pill_list()` creates a vertical list of navigation controls adjadent to, rather than on top of, the tab content panels.

```
navset_pill_list(
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```



`page_navbar()`:

Finally, `page_navbar()` provides full-page navigation container similar to `navset_underline()` but where each `nav_panel()` is treated as a full page of content and the navigation controls appear in a top-level navigation bar. Note that the navbar background and underline styling can be controlled via `navbar_options`.

`page_navbar()` is complimented by `navset_bar()` which produces a similar layout intended to be used within an app.

```
page_navbar(
  title = "My App",
```

```
  navbar_options = navbar_options(
    bg = "#0062cc",
    underline = TRUE
  ),
  nav_panel(title = "One", p("First tab content.")),
  nav_panel(title = "Two", p("Second tab content.")),
  nav_panel(title = "Three", p("Third tab content")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    align = "right",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)
```

See Also

nav_panel(), nav_panel_hidden() create panels of content.

nav_menu(), nav_item(), nav_spacer() create menus, items, or space in the navset control area.

nav_insert(), nav_remove() programmatically add or remove nav panels.

nav_select(), nav_show(), nav_hide() change the state of a nav_panel() in a navset.

Other Panel container functions: nav-items, nav_select()

---

nav_select *Dynamically update nav containers*

Description

Functions for dynamically updating nav containers (e.g., select, insert, and remove nav items). These functions require an `id` on the nav container to be specified and must be called within an active Shiny session.

Usage

```
nav_select(id, selected = NULL, session = get_current_session())

nav_insert(
  id,
  nav,
  target = NULL,
  position = c("after", "before"),
  select = FALSE,
  session = get_current_session()
)

nav_remove(id, target, session = get_current_session())

nav_show(id, target, select = FALSE, session = get_current_session())

nav_hide(id, target, session = get_current_session())
```

Arguments

id	a character string used to identify the nav container.
selected	a character string used to identify a particular nav_panel() item.
session	a shiny session object (the default should almost always be used).
nav	a nav_panel() item.
target	The value of an existing nav_panel() item, next to which tab will be added. If removing: the value of the nav_panel() item that you want to remove.
position	Should nav be added before or after the target?
select	Should nav be selected upon being inserted?

See Also

Navset functions create the navigation container holding the nav panels.

nav_panel(), nav_panel_hidden() create panels of content.

nav_menu(), nav_item(), nav_spacer() create menus, items, or space in the navset control area.

Other Panel container functions: nav-items, navset

Examples

```
can_browse <- function() rlang::is_interactive() && require("shiny")

# Selecting a tab
if (can_browse()) {
  shinyApp(
    page_fluid(
      radioButtons("item", "Choose", c("A", "B")),
      navset_hidden(
        id = "container",
```

```
      nav_panel_hidden("A", "a"),
      nav_panel_hidden("B", "b")
    )
  ),
  function(input, output) {
    observe(nav_select("container", input$item))
  }
 )
}

# Inserting and removing
if (can_browse()) {
  ui <- page_fluid(
    actionButton("add", "Add 'Dynamic' tab"),
    actionButton("remove", "Remove 'Foo' tab"),
    navset_tab(
      id = "tabs",
      nav_panel("Hello", "hello"),
      nav_panel("Foo", "foo"),
      nav_panel("Bar", "bar tab")
    )
  )
  server <- function(input, output) {
    observeEvent(input$add, {
      nav_insert(
        "tabs", target = "Bar", select = TRUE,
        nav_panel("Dynamic", "Dynamically added content")
      )
    })
    observeEvent(input$remove, {
      nav_remove("tabs", target = "Foo")
    })
  }
  shinyApp(ui, server)
}
```

---

page *Modern Bootstrap page layouts*

Description

These functions are small wrappers around shiny's page constructors (i.e., `shiny::fluidPage()`, `shiny::navbarPage()`, etc) that differ in two ways:

- The `theme` parameter defaults bslib's recommended version of Bootstrap (for new projects).
- The return value is rendered as an static HTML page when printed interactively at the console.

Usage

```
page(..., title = NULL, theme = bs_theme(), lang = NULL)

page_fluid(..., title = NULL, theme = bs_theme(), lang = NULL)

page_fixed(..., title = NULL, theme = bs_theme(), lang = NULL)
```

Arguments

...	UI elements for the page. Named arguments become HTML attributes.
title	The browser window title (defaults to the host URL of the page)
theme	A bs_theme() object.
lang	ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

Functions

- page(): A **bslib** wrapper for shiny::bootstrapPage(), a basic Boostrap page where the content is added directly to the page body.
- page_fluid(): A **bslib** wrapper for shiny::fluidPage(), a fluid Bootstrap-based page layout that extends to the full viewport width.
- page_fixed(): A **bslib** wrapper for shiny::fixedPage(), a fixed Bootstrap-based page layout where the page content container is centered horizontally and its width is constrained.

See Also

Dashboard-style pages: page_sidebar(), page_navbar(), page_fillable().

---

page_fillable *A screen-filling page layout*

Description

A Bootstrap-based page layout whose contents fill the full height and width of the browser window.

Usage

```
page_fillable(
  ...,
  padding = NULL,
  gap = NULL,
  fillable_mobile = FALSE,
  title = NULL,
  theme = bs_theme(),
  lang = NULL
)
```

Arguments

...	UI elements for the page. Named arguments become HTML attributes.
padding	Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.
gap	A CSS length unit defining the gap (i.e., spacing) between elements provided to ....
fillable_mobile	Whether or not the page should fill the viewport's height on mobile devices (i.e., narrow windows).
title	The browser window title (defaults to the host URL of the page)
theme	A bs_theme() object.
lang	ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

References

- Filling Layouts on the bslib website.
- Getting Started with Dashboards on the bslib website.

See Also

layout_columns() and layout_column_wrap() for laying out content into rows and columns.

layout_sidebar() for 'floating' sidebar layouts.

accordion() for grouping related input controls in the sidebar.

card() for wrapping outputs in the 'main' content area.

value_box() for highlighting values.

Other Dashboard page layouts: page_navbar(), page_sidebar()

Examples

```
library(shiny)
library(ggplot2)
library(bslib)

ui <- page_fillable(
  h1("Example", code("mtcars"), "dashboard"),
  layout_columns(
    card(
      full_screen = TRUE,
```

```
      card_header("Number of forward gears"),
      plotOutput("gear")
    ),
    card(
      full_screen = TRUE,
      card_header("Number of carburetors"),
      plotOutput("carb")
    )
  ),
  card(
    full_screen = TRUE,
    card_header("Weight vs. Quarter Mile Time"),
    layout_sidebar(
      sidebar = sidebar(
        varSelectInput("var_x", "Compare to qsec:", mtcars[-7], "wt"),
        varSelectInput("color", "Color by:", mtcars[-7], "cyl"),
        position = "right"
      ),
      plotOutput("var_vs_qsec")
    )
  )
)

server <- function(input, output) {
  for (var in c("cyl", "vs", "am", "gear", "carb")) {
    mtcars[[var]] <- as.factor(mtcars[[var]])
  }

  output$gear <- renderPlot({
    ggplot(mtcars, aes(gear)) + geom_bar()
  })

  output$carb <- renderPlot({
    ggplot(mtcars, aes(carb)) + geom_bar()
  })

  output$var_vs_qsec <- renderPlot({
    req(input$var_x, input$color)

    ggplot(mtcars) +
      aes(y = qsec, x = !!input$var_x, color = !!input$color) +
      geom_point()
  })
}

shinyApp(ui, server)
```

---

page_navbar *Multi-page app with a top navigation bar*

Description

Create a page that contains a top level navigation bar that can be used to toggle a set of nav_panel() elements. Use this page layout to create the effect of a multi-page app, where your app's content is broken up into multiple "pages" that can be navigated to via the top navigation bar.

Usage

```
page_navbar(
  ...,
  title = NULL,
  id = NULL,
  selected = NULL,
  sidebar = NULL,
  fillable = TRUE,
  fillable_mobile = FALSE,
  gap = NULL,
  padding = NULL,
  header = NULL,
  footer = NULL,
  navbar_options = NULL,
  fluid = TRUE,
  theme = bs_theme(),
  window_title = NA,
  lang = NULL,
  position = deprecated(),
  bg = deprecated(),
  inverse = deprecated(),
  underline = deprecated(),
  collapsible = deprecated()
)
```

Arguments

...	a collection of nav_panel() items.
title	A (left-aligned) title to place in the card header/footer. If provided, other nav items are automatically right aligned.
id	a character string used for dynamically updating the container (see nav_select()).
selected	a character string matching the value of a particular nav_panel() item to selected by default.
sidebar	A sidebar() component to display on every nav_panel() page.
fillable	Whether or not to allow fill items to grow/shrink to fit the browser window. If TRUE, all nav_panel() pages are fillable. A character vector, matching the value of nav_panel()s to be filled, may also be provided. Note that, if a sidebar is provided, fillable makes the main content portion fillable.
fillable_mobile	Whether or not fillable pages should fill the viewport's height on mobile devices (i.e., narrow windows).

gap	A CSS length unit defining the gap (i.e., spacing) between elements provided to ....
padding	Padding to use for the body. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. The length can be between one and four. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.
header	UI element(s) (htmltools::tags) to display *above* the nav content. For `card`-based navsets, these elements are implicitly wrapped in a `card_body()`. To control things like `padding`, `fill`, etc., wrap the elements in an explicit `card_body()`.
footer	UI element(s) (htmltools::tags) to display *below* the nav content. For `card`-based navsets, these elements are implicitly wrapped in a `card_body()`. To control things like `padding`, `fill`, etc., wrap the elements in an explicit `card_body()`.
navbar_options	Options to control the appearance and behavior of the navbar. Use `navbar_options()` to create the list of options.
fluid	`TRUE` to use fluid layout; `FALSE` to use fixed layout.
theme	A `bs_theme()` object.
window_title	the browser window title. The default value, `NA`, means to use any character strings that appear in `title` (if none are found, the host URL of the page is displayed by default).
lang	ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the `<html>` tag, as in `<html lang="en">`. The default (NULL) results in an empty string.
position	**[Deprecated]** Please use `navbar_options = navbar_options(position=)` instead.
bg	**[Deprecated]** Please use `navbar_options = navbar_options(bg=)` instead.
inverse	**[Deprecated]** Please use `navbar_options = navbar_options(inverse=)` instead.
underline	**[Deprecated]** Please use `navbar_options = navbar_options(underline=)` instead.
collapsible	**[Deprecated]** Please use `navbar_options = navbar_options(collapsible=)` instead.

References

Getting Started with Dashboards on the bslib website.

See Also

`nav_panel()`, `nav_menu()`, and `nav_item()` for adding content sections and organizing or creating items in the navigation bar.

`layout_columns()` and `layout_column_wrap()` for laying out content into rows and columns.

card() for wrapping outputs in the 'main' content area.

value_box() for highlighting values.

accordion() for grouping related input controls in the sidebar.

Other Dashboard page layouts: page_fillable(), page_sidebar()

Examples

```
library(shiny)
library(bslib)

link_shiny <- tags$a(
  shiny::icon("github"), "Shiny",
  href = "https://github.com/rstudio/shiny",
  target = "_blank"
)
link_posit <- tags$a(
  shiny::icon("r-project"), "Posit",
  href = "https://posit.co",
  target = "_blank"
)

ui <- page_navbar(
  title = "My App",
  nav_panel(title = "One", p("First page content.")),
  nav_panel(title = "Two", p("Second page content.")),
  nav_panel("Three", p("Third page content.")),
  nav_spacer(),
  nav_menu(
    title = "Links",
    align = "right",
    nav_item(link_shiny),
    nav_item(link_posit)
  )
)

server <- function(...) { } # not used in this example

shinyApp(ui, server)
```

---

page_sidebar *A sidebar page (i.e., dashboard)*

Description

Create a dashboard layout with a full-width header (title) and sidebar().

Usage

```
page_sidebar(
  ...,
  sidebar = NULL,
  title = NULL,
  fillable = TRUE,
  fillable_mobile = FALSE,
  theme = bs_theme(),
  window_title = NA,
  lang = NULL
)
```

Arguments

...	UI elements to display in the 'main' content area (i.e., next to the sidebar). These arguments are passed to layout_sidebar(), which has more details.
sidebar	A sidebar() object.
title	A string, number, or htmltools::tag() child to display as the title (just above the sidebar).
fillable	Whether or not the main content area should be considered a fillable (i.e., flexbox) container.
fillable_mobile	Whether or not the page should fill the viewport's height on mobile devices (i.e., narrow windows).
theme	A bs_theme() object.
window_title	the browser window title. The default value, NA, means to use any character strings that appear in title (if none are found, the host URL of the page is displayed by default).
lang	ISO 639-1 language code for the HTML page, such as "en" or "ko". This will be used as the lang in the <html> tag, as in <html lang="en">. The default (NULL) results in an empty string.

References

Getting Started with Dashboards on the bslib website.

See Also

layout_columns() and layout_column_wrap() for laying out content into rows and columns.

accordion() for grouping related input controls in the sidebar.

card() for wrapping outputs in the 'main' content area.

value_box() for highlighting values.

Other Dashboard page layouts: page_fillable(), page_navbar()

Examples

```
library(shiny)
library(ggplot2)
library(bslib)

ui <- page_sidebar(
  title = "Example dashboard",
  sidebar = sidebar(
    varSelectInput("var", "Select variable", mtcars)
  ),
  card(
    full_screen = TRUE,
    card_header("My plot"),
    plotOutput("p")
  )
)

server <- function(input, output) {
  output$p <- renderPlot({
    ggplot(mtcars) + geom_histogram(aes(!!input$var))
  })
}

shinyApp(ui, server)
```

---

popover *Add a popover to a UI element*

Description

Display additional information when clicking on a UI element (typically a button).

Usage

```
popover(
  trigger,
  ...,
  title = NULL,
  id = NULL,
  placement = c("auto", "top", "right", "bottom", "left"),
  options = list()
)

toggle_popover(id, show = NULL, session = get_current_session())

update_popover(id, ..., title = NULL, session = get_current_session())
```

Arguments

trigger	The UI element to serve as the popover trigger (typically a shiny::actionButton() or similar). If trigger renders as multiple HTML elements (e.g., it's a tagList()), the last HTML element is used for the trigger. If the trigger should contain all of those elements, wrap the object in a htmltools::div() or htmltools::span().
...	UI elements for the popover's body. Character strings are automatically escaped unless marked as htmltools::HTML().
title	A title (header) for the popover. To remove a header with update_popover(), provide a either an empty string or character(0).
id	A character string. Required to re-actively respond to the visibility of the popover (via the input[[id]] value) and/or update the visibility/contents of the popover.
placement	The placement of the popover relative to its trigger.
options	A list of additional options.
show	Whether to show (TRUE) or hide (FALSE) the popover. The default (NULL) will show if currently hidden and hide if currently shown. Note that a popover will not be shown if the trigger is not visible (e.g., it's hidden behind a tab).
session	A Shiny session object (the default should almost always be used).

Functions

- popover(): Add a popover to a UI element
- toggle_popover(): Programmatically show/hide a popover.
- update_popover(): Update the contents of a popover.

Closing popovers

In addition to clicking the close_button, popovers can be closed by pressing the Esc/Space key when the popover (and/or its trigger) is focused.

Theming/Styling

Like other bslib components, popovers can be themed by supplying relevant theming variables to bs_theme(), which effects styling of every popover on the page. To style a *specific* popover differently from other popover, utilize the customClass option:

```
popover(
  "Trigger", "Popover message",
  options = list(customClass = "my-pop")
)
```

And then add relevant rules to bs_theme() via bs_add_rules():

```
bs_theme() |> bs_add_rules(".my-pop { max-width: none; }")
```

Accessibility of Popover Triggers

Because the user needs to interact with the `trigger` element to see the popover, it's best practice to use an element that is typically accessible via keyboard interactions, like a button or a link. If you use a non-interactive element, like a `<span>` or text, bslib will automatically add the `tabindex="0"` attribute to the trigger element to make sure that users can reach the element with the keyboard. This means that in most cases you can use any element you want as the trigger.

One place where it's important to consider the accessibility of the trigger is when using an icon without any accompanying text. In these cases, many R packages that provide icons will create an icon element with the assumption that the icon is decorative, which will make it inaccessible to users of assistive technologies.

When using an icon as the primary trigger, ensure that the icon does not have `aria-hidden="true"` or `role="presentation"` attributes. Icon packages typically provide a way to specify a title for the icon, as well as a way to specify that the icon is not decorative. The title should be a short description of the purpose of the trigger, rather than a description of the icon itself.

- If you're using `bsicons::bs_icon()`, provide a `title`.
- If you're using `fontawesome::fa()`, set `a11y = "sem"` and provide a `title`.

For example:

```
popover(
  bsicons::bs_icon("gear", title = "Settings"),
  title = "Settings",
  sliderInput("n", "Number of points", 1, 100, 50)
)

popover(
  fontawesome::fa("gear", a11y = "sem", title = "Settings"),
  title = "Settings",
  sliderInput("n", "Number of points", 1, 100, 50)
)
```

References

Popovers are based on Bootstrap's Popover component. See the bslib website for an interactive introduction to tooltips and popovers.

See Also

`tooltip()` provides an alternative way to display informational text on demand, typically when focusing or hovering over a trigger element.

Other Components: `accordion()`, `card()`, `tooltip()`, `value_box()`

Examples

```
popover(
  shiny::actionButton("btn", "A button"),
```

```
  "Popover body content...",
  title = "Popover title"
)

library(shiny)
library(bslib)

ui <- page_fixed(
  card(class = "mt-5",
    card_header(
      popover(
        uiOutput("card_title", inline = TRUE),
        title = "Provide a new title",
        textInput("card_title", NULL, "An editable title")
      )
    ),
    "The card body..."
  )
)

server <- function(input, output) {
  output$card_title <- renderUI({
    list(input$card_title, bsicons::bs_icon("pencil-square"))
  })
}

shinyApp(ui, server)
```

`run_with_themer` *Theme customization UI*

Description

A 'real-time' theme customization UI that you can use to easily make common tweaks to Bootstrap variables and immediately see how they would affect your app's appearance. There are two ways you can launch the theming UI. For most Shiny apps, just use `run_with_themer()` in place of `shiny::runApp()`; they should take the same arguments and work the same way. Alternatively, you can call the `bs_themer()` function from inside your server function (or in an R Markdown app that is using `runtime: shiny`, you can call this from any code chunk). Note that this function is only intended to be used for development!

Usage

```
run_with_themer(appDir = getwd(), ..., gfonts = TRUE, gfonts_update = FALSE)

bs_themer(gfonts = TRUE, gfonts_update = FALSE)
```

Arguments

appDir	The application to run. This can be a file or directory path, or a shiny::shinyApp() object. See shiny::runApp() for details.
...	Additional parameters to pass through to shiny::runApp().
gfonts	whether or not to detect Google Fonts and wrap them in font_google() (so that their font files are automatically imported).
gfonts_update	whether or not to update the internal database of Google Fonts.

Details

To help you utilize the changes you see in the preview, this utility prints bs_theme() code to the R console.

Value

nothing. These functions are called for their side-effects.

Limitations

- Doesn't work with Bootstrap 3.
- Doesn't work with IE11.
- Only works inside Shiny apps and `runtime: shiny` R Markdown documents.
  - Can't be used with static R Markdown documents.
  - Can be used to some extent with `runtime: shiny_prerendered`, but only UI rendered through a `context="server"` may update in real-time.
- Doesn't work with '3rd party' custom widgets that don't make use of bs_dependency_defer() or bs_current_theme().

Examples

```
library(shiny)
library(bslib)

ui <- fluidPage(
  theme = bs_theme(bg = "black", fg = "white"),
  h1("Heading 1"),
  h2("Heading 2"),
  p(
    "Paragraph text;",
    tags$a(href = "https://www.rstudio.com", "a link")
  ),
  p(
    actionButton("cancel", "Cancel"),
    actionButton("continue", "Continue", class = "btn-primary")
  ),
  tabsetPanel(
    tabPanel("First tab",
      "The contents of the first tab"
```

```
    ),
    tabPanel("Second tab",
      "The contents of the second tab"
    )
  )
)

if (interactive()) {
  run_with_themer(shinyApp(ui, function(input, output) {}))
}
```

`show_toast` *Show or hide a toast notification*

Description

Displays a toast notification in a Shiny application.

Usage

```
show_toast(toast, ..., session = shiny::getDefaultReactiveDomain())

hide_toast(id, ..., session = shiny::getDefaultReactiveDomain())
```

Arguments

`toast`	A `toast()`, or a string that will be automatically converted to a toast with default settings.
`...`	Reserved for future extensions (currently ignored).
`session`	Shiny session object.
`id`	String with the toast ID returned by `show_toast()` or a `toast` object provided that the `id` was set when created/shown.

Value

`show_toast()` Invisibly returns the toast ID (string) that can be used with `hide_toast()`.

Functions

- `show_toast()`: Show a toast notification.
- `hide_toast()`: Hide a toast notification by ID.

See Also

Other Toast components: `toast()`

Examples

```
library(shiny)
library(bslib)

ui <- page_fluid(
  actionButton("show_persistent", "Show Persistent Toast"),
  actionButton("hide_persistent", "Hide Toast")
)

server <- function(input, output, session) {
  toast_id <- reactiveVal(NULL)

  observeEvent(input$show_persistent, {
    id <- show_toast(
      toast(
        body = "This toast won't disappear automatically.",
        autohide = FALSE
      )
    )
    toast_id(id)
  })

  observeEvent(input$hide_persistent, {
    req(toast_id())
    hide_toast(toast_id())
    toast_id(NULL)
  })
}

shinyApp(ui, server)
```

---

sidebar *Sidebar layouts*

Description

Sidebar layouts place UI elements, like input controls or additional context, next to the main content area which often holds output elements like plots or tables.

There are several page, navigation, and layout functions that allow you to create a sidebar layout. In each case, you can create a collapsing sidebar layout by providing a `sidebar()` object to the `sidebar` argument the following functions.

- `page_sidebar()` creates a "page-level" sidebar.
- `page_navbar()` creates a multi-panel app with an (optional, page-level) sidebar that is shown on every panel.
- `layout_sidebar()` creates a "floating" sidebar layout component which can be used inside any `page()` and/or `card()` context.

- navset_card_tab() and navset_card_pill() create multi-tab cards with a shared sidebar that is accessible from every panel.

See the Sidebars article on the bslib website to learn more.

Usage

```
sidebar(
  ...,
  width = 250,
  position = c("left", "right"),
  open = NULL,
  id = NULL,
  title = NULL,
  bg = NULL,
  fg = NULL,
  class = NULL,
  max_height_mobile = NULL,
  gap = NULL,
  padding = NULL,
  fillable = FALSE,
  resizable = TRUE
)

layout_sidebar(
  ...,
  sidebar = NULL,
  fillable = TRUE,
  fill = TRUE,
  bg = NULL,
  fg = NULL,
  border = NULL,
  border_radius = NULL,
  border_color = NULL,
  padding = NULL,
  gap = NULL,
  height = NULL
)

toggle_sidebar(id, open = NULL, session = get_current_session())
```

Arguments

...	Unnamed arguments can be any valid child of an htmltools tag and named arguments become HTML attributes on returned UI element. In the case of layout_sidebar(), these arguments are passed to the main content tag (not the sidebar+main content container).
width	A valid CSS unit used for the width of the sidebar.
position	Where the sidebar should appear relative to the main content.

open
: The initial state of the sidebar, choosing from the following options:
  - "desktop": The sidebar starts open on desktop screen, closed on mobile. This is default sidebar behavior.
  - "open" or TRUE: The sidebar starts open.
  - "closed" or FALSE: The sidebar starts closed.
  - "always" or NA: The sidebar is always open and cannot be closed.

  Alternatively, you can use a list with desktop or mobile items to set the initial sidebar state independently for desktop and mobile screen sizes. In this case, desktop or mobile can use any of the above options except "desktop", which is equivalent to list(desktop = "open", mobile = "closed"). You can also choose to place an always open sidebar above the main content on mobile devices by setting mobile = "always-above".

  In sidebar_toggle(), open indicates the desired state of the sidebar, where the default of open = NULL will cause the sidebar to be toggled open if closed or vice versa. Note that sidebar_toggle() can only open or close the sidebar, so it does not support the "desktop" and "always" options.

id
: A character string. Required if wanting to re-actively read (or update) the collapsible state in a Shiny app.

title
: A character title to be used as the sidebar title, which will be wrapped in a <header> element with class sidebar-title. You can also provide a custom htmltools::tag() for the title element, in which case you'll likely want to give this element class = "sidebar-title".

bg, fg
: A background or foreground color. If only one of either is provided, an accessible contrasting color is provided for the opposite color, e.g. setting bg chooses an appropriate fg color.

class
: CSS classes for the sidebar container element, in addition to the fixed .sidebar class.

max_height_mobile
: A CSS length unit defining the maximum height of the horizontal sidebar when viewed on mobile devices. Only applies to always-open sidebars that use open = "always", where by default the sidebar container is placed below the main content container on mobile devices.

gap
: A CSS length unit defining the vertical gap (i.e., spacing) between adjacent elements provided to ....

padding
: Padding within the sidebar itself. This can be a numeric vector (which will be interpreted as pixels) or a character vector with valid CSS lengths. padding may be one to four values. If one, then that value will be used for all four sides. If two, then the first value will be used for the top and bottom, while the second value will be used for left and right. If three, then the first will be used for top, the second will be left and right, and the third will be bottom. If four, then the values will be interpreted as top, right, bottom, and left respectively.

fillable
: Whether or not the main content area should be considered a fillable (i.e., flexbox) container.

resizable
: Whether the sidebar can be resized by dragging its edge. When TRUE (the default), a resize handle is added to the sidebar that allows users to adjust the sidebar width on desktop (wide screen sizes).

sidebar	A sidebar() object.
fill	Whether or not to allow the layout container to grow/shrink to fit a fillable container with an opinionated height (e.g., page_fillable()).
border	Whether or not to add a border.
border_radius	Whether or not to add a border radius.
border_color	The border color that is applied to the entire layout (if border = TRUE) and the color of the border between the sidebar and the main content area.
height	Any valid CSS unit (e.g., height="200px"). Doesn't apply when a card is made full_screen (in this case, consider setting a height in card_body()).
session	A Shiny session object (the default should almost always be used).

Functions

- toggle_sidebar(): Toggle a sidebar() state during an active Shiny user session. To use this function, the sidebar() you want to open or close must have an id value.

References

Sidebar layouts are featured in a number of pages on the bslib website:

- Sidebars
- Cards: Sidebars
- Getting Started: Dashboards

---

theme_bootswatch *Obtain a theme's Bootswatch theme name*

Description

Obtain a theme's Bootswatch theme name

Usage

```
theme_bootswatch(theme)
```

Arguments

theme	A bs_theme() object.

Value

Returns the Bootswatch theme named used (if any) in the theme.

See Also

Other Bootstrap theme utility functions: bootswatch_themes(), bs_get_variables(), builtin_themes(), theme_version(), versions()

theme_version *Obtain a theme's Bootstrap version*

Description

Obtain a theme's Bootstrap version

Usage

```
theme_version(theme)
```

Arguments

theme	A bs_theme() object.

Value

Returns the major version of Bootstrap used in the theme.

See Also

Other Bootstrap theme utility functions: bootswatch_themes(), bs_get_variables(), builtin_themes(), theme_bootswatch(), versions()

toast *Toast notifications*

Description

Toast notifications are lightweight, temporary messages designed to mimic push notifications from mobile and desktop operating systems. They are built on Bootstrap 5's toast component.

bslib includes a complete example of toasts and their many configuration options:

```
shiny::runExample("toast", package = "bslib")
```

Usage

```
toast(
  ...,
  header = NULL,
  icon = NULL,
  id = NULL,
  type = NULL,
  duration_s = 5,
  position = "top-right",
```

```
  closable = TRUE
)

toast_header(title, ..., icon = NULL, status = NULL)
```

Arguments

`...`	Body content of the toast. Can be a string, or any HTML elements. Named arguments will be treated as HTML attributes for the toast container.
`header`	Optional header content. Can be a string, or the result of `toast_header()`. If provided, creates a `.toast-header` with close button (if `closable = TRUE`).
`icon`	Optional icon element, for example from `shiny::icon()`, `bsicons::bs_icon()` or `fontawesome::fa()`.
`id`	Optional unique identifier for the toast. If `NULL`, an ID will be automatically generated when the toast is shown via `show_toast()`. Providing a stable ID allows you to hide the toast later. If a toast with `id` is already visible, that toast is automatically hidden before showing the new toast with the same `id` so that only one toast with a given ID is shown at once.
`type`	Optional semantic type. One of `NULL`, `"primary"`, `"secondary"`, `"success"`, `"info"`, `"warning"`, `"danger"`, `"light"`, or `"dark"`. Applies appropriate Bootstrap background utility classes (`text-bg-*`).
`duration_s`	Numeric. Number of seconds after which the toast should automatically hide. Use `0`, or `NA` to disable auto-hiding (toast will remain visible until manually dismissed). Default is `5` (5 seconds).
`position`	String or character vector specifying where to position the toast container. Can be provided in several formats: • Kebab-case: `"top-left"`, `"bottom-right"`, etc. • Space-separated: `"top left"`, `"bottom right"`, etc. • Character vector: `c("top", "left")`, `c("bottom", "right")`, etc. • Any order: `"left top"` is equivalent to `"top left"` Valid vertical positions are `"top"`, `"middle"`, or `"bottom"`. Valid horizontal positions are `"left"`, `"center"`, or `"right"`. Input is case-insensitive. Default is `"bottom-right"`.
`closable`	Logical. Whether to include a close button. Defaults to `TRUE`. When both `duration_s = NA` (or `0` or `NULL`) and `closable = FALSE`, the toast will remain visible until manually hidden via `hide_toast()`. This is useful when the toast contains interactive Shiny UI elements and you want to manage the toast display programmatically.
`title`	Header text (required).
`status`	Optional status text that appears as small, muted text on the right side of the header.

Value

A `bslib_toast` object that can be passed to `show_toast()`.

For `toast_header()`: a toast header object that can be used with the `header` argument of `toast()`.

Functions

- `toast()`: Create a toast element.
- `toast_header()`: Create a structured toast header with optional icon and status indicator. Returns a data structure that can be passed to the `header` argument of `toast()`.

See Also

`show_toast()` to display a toast, `hide_toast()` to dismiss a toast, and `toast_header()` to create structured headers.

Other Toast components: `show_toast()`

Examples

```
library(shiny)
library(bslib)

ui <- page_fluid(
  actionButton("show_simple", "Simple Toast"),
  actionButton("show_header", "Toast with Header")
)

server <- function(input, output, session) {
  observeEvent(input$show_simple, {
    show_toast(
      toast(
        "Operation completed successfully!",
        header = "Success",
        type = "success"
      )
    )
  })

  observeEvent(input$show_header, {
    show_toast(
      toast(
        "Your settings have been saved.",
        header = toast_header(
          title = "Settings Updated",
          status = "just now"
        ),
        type = "success"
      )
    )
  })
}

shinyApp(ui, server)
```

`toolbar` *Toolbar component*

Description

A toolbar which can contain buttons, inputs, and other UI elements in a small form suitable for inclusion in card headers, footers, and other small places.

bslib includes a complete example of toolbars and the many ways they can be used:

```
shiny::runExample("toolbar", package = "bslib")
```

Usage

```
toolbar(..., align = c("right", "left"), gap = NULL, width = NULL)
```

Arguments

`...`	UI elements for the toolbar.
`align`	Determines if toolbar should be aligned to the `"right"` or `"left"`.
`gap`	A CSS length unit defining the gap (i.e., spacing) between elements in the toolbar. Defaults to `0` (no gap).
`width`	CSS width of the toolbar. Defaults to `NULL`, which will automatically set `width: 100%` when the toolbar is a direct child of a label element (e.g., when used in input labels). For `toolbar_spacer()` to push elements effectively, the toolbar needs `width: 100%` to expand and create space. Set this explicitly if you need to control the width in other contexts.

Value

Returns a toolbar element.

Cookbook

Toolbars in Card Headers and Footers:

Card headers and footers are a common places you might want to use toolbars. Toolbars allow you to clearly show that a selection of inputs pertain to that particular card. For example, this card uses multiple `toolbar_input_select()` for filtering and sorting, along with toolbar buttons and a "share" button in the toolbar in the card footer.

```
card(
  full_screen = TRUE,
  card_header(
    "Sales Data",
    toolbar(
      align = "right",
      toolbar_input_select(
```

```
          id = "filter",
          label = "Filter",
          choices = c("All", "Active", "Inactive"),
          icon = icon("filter")
        ),
        toolbar_input_select(
          id = "sort",
          label = "Sort by",
          choices = c("Date", "Amount", "Customer"),
          icon = icon("sort")
        ),
        toolbar_divider(),
        toolbar_input_button(
          id = "refresh",
          label = "Refresh",
          icon = icon("arrows-rotate")
        ),
        toolbar_input_button(
          id = "export",
          label = "Export",
          icon = icon("download"),
          show_label = TRUE
        )
      )
    ),
    card_body(
      h3("Card Body Here"),
    ),
    card_footer(
      toolbar(
        align = "right",
        toolbar_input_button(
          id = "share_data",
          label = "Share",
          icon = icon("share-nodes"),
          show_label = TRUE,
          border = TRUE
        )
      )
    )
  )
```

Toolbars in labels:

You can use toolbars in the labels of other Shiny inputs to add a composite input with additional buttons or controls. The following example uses a `toolbar()` in the label of a `shiny::numericInput()` to add increment and decrement buttons next to the label:

```
shiny::numericInput(
  "quantity",
```

```
  label = toolbar(
    "Quantity",
    toolbar_spacer(), # push buttons to the right
    toolbar_input_button("decrement", "Less", icon("minus")),
    toolbar_input_button("increment", "More", icon("plus"))
  ),
  value = 1,
  min = 0,
  max = 100
)
```

You can also use toolbars in the labels of text area inputs. For example, this text editor uses a toolbar with formatting buttons:

```
textAreaInput(
  "editor",
  label = toolbar(
    "Comment",
    toolbar_spacer(),
    toolbar_input_button("bold", label = "Bold", icon = icon("bold")),
    toolbar_input_button("italic", label = "Italic", icon = icon("italic")),
    toolbar_input_button("link", label = "Link", icon = icon("link"))
  ),
  value = "",
  rows = 5,
  placeholder = "Type your comment here..."
)
```

The input_submit_textarea() function from bslib allows you to create a text area input with a submit button and an optional toolbar. Here is an example of using a toolbar with formatting buttons and options:

```
input_submit_textarea(
  "message",
  placeholder = "Type a message...",
  toolbar = toolbar(
    toolbar_input_button("attach", icon = icon("paperclip"), label = "Attach"),
    toolbar_input_button("emoji", icon = icon("face-smile"), label = "Emoji"),
    toolbar_divider(),
    toolbar_input_select(
      "format",
      label = "Format",
      choices = c("Plain", "Markdown", "HTML"),
      icon = icon("code")
    )
  )
)
```

See Also

card_header() for using toolbars in card headers/footers

Other toolbar components: toolbar_divider(), toolbar_input_button(), toolbar_input_select()

Examples

```
# Minimal toolbar example
toolbar(
  toolbar_input_button(id = "view", icon = icon("eye"), label = "View"),
  toolbar_input_button(id = "save", icon = icon("save"), label = "Save"),
  toolbar_divider(),
  toolbar_input_select(
    id = "filter",
    label = "Filter",
    choices = c("All", "Active", "Inactive")
  )
)

# Toolbar with text input
library(shiny)
library(bslib)

ui <- page_fluid(
  numericInput(
    "quantity",
    label = toolbar(
      "Quantity",
      toolbar_spacer(),
      toolbar_input_button("decrement", "Less", icon("minus")),
      toolbar_input_button("increment", "More", icon("plus"))
    ),
    value = 5,
    min = 0,
    max = 100
  ),
  verbatimTextOutput("value")
)

server <- function(input, output, session) {
  output$value <- renderText({
    paste("Current value:", input$quantity)
  })

  observeEvent(input$increment, {
    updateNumericInput(session, "quantity", value = input$quantity + 1)
  })

  observeEvent(input$decrement, {
    updateNumericInput(session, "quantity", value = input$quantity - 1)
  })
}

shinyApp(ui, server)

# Toolbar with input_submit_textarea()
```

```
library(shiny)
library(bslib)

ui <- page_fluid(
  input_submit_textarea(
    "message",
    placeholder = "Type a message...",
    toolbar = toolbar(
      toolbar_input_button("attach", icon = icon("paperclip"), label = "Attach"),
      toolbar_input_button("emoji", icon = icon("face-smile"), label = "Emoji"),
      toolbar_divider(),
      toolbar_input_select(
        "format",
        label = "Format",
        choices = c("Plain", "Markdown", "HTML"),
        icon = icon("code")
      )
    )
  ),
  verbatimTextOutput("output")
)

server <- function(input, output, session) {
  output$output <- renderText({
    req(input$message)
    paste("You said:", input$message)
  })

  observeEvent(input$attach, {
    showNotification("Attach clicked!", duration = 2)
  })

  observeEvent(input$emoji, {
    showNotification("Emoji clicked!", duration = 2)
  })
}

shinyApp(ui, server)
```

`toolbar_divider` *Toolbar: Add a divider or spacer to a toolbar*

Description

`toolbar_divider()` creates a visual divider line with customizable and fixed width and spacing between toolbar elements. `toolbar_spacer()` creates empty space that expands to push adjacent toolbar elements apart as much as possible.

Usage

```
toolbar_divider(..., width = NULL, gap = NULL)

toolbar_spacer()
```

Arguments

`...`	Ignored, reserved for future use and to require named arguments in `toolbar_divider()`.
`width`	A CSS length unit specifying the width of the divider line. Defaults to `"2px"` for a sensible dividing line. Pass `0px` for no divider line.
`gap`	A CSS length unit defining the spacing around the divider. Defaults to `"1rem"` for sensible fixed spacing.

Functions

- `toolbar_divider()`: Create a dividing line and fixed space between toolbar elements.
- `toolbar_spacer()`: Create empty, expanding space between toolbar elements. Note that for the spacer to push elements effectively, the parent toolbar needs `width: 100%` (which is automatically applied when the toolbar is a direct child of a label element). If the spacer doesn't appear to work, you may need to set `width = "100%"` explicitly on the `toolbar()`.

See Also

Other toolbar components: `toolbar()`, `toolbar_input_button()`, `toolbar_input_select()`

Examples

```
toolbar(
  toolbar_input_button(id = "left1", label = "Left"),
  toolbar_divider(),
  toolbar_input_button(id = "right1", label = "Right")
)

toolbar(
  toolbar_input_button(id = "a", label = "A"),
  toolbar_divider(width = "5px", gap = "20px"),
  toolbar_input_button(id = "b", label = "B")
)

toolbar(
  toolbar_input_button(id = "previous", label = "Previous"),
  toolbar_spacer(),
  toolbar_input_button(id = "next", label = "Next")
)
```

`toolbar_input_button` *Add a toolbar input button*

Description

A button designed to fit well in small places such as in a `toolbar()`.

Usage

```
toolbar_input_button(
  id,
  label,
  icon = NULL,
  show_label = is.null(icon),
  tooltip = !show_label,
  ...,
  disabled = FALSE,
  border = FALSE
)

update_toolbar_input_button(
  id,
  label = NULL,
  show_label = NULL,
  icon = NULL,
  disabled = NULL,
  session = get_current_session()
)
```

Arguments

`id`	The input ID.
`label`	The input label. By default, `label` is not shown but is used by `tooltip`. Set `show_label = TRUE` to show the label (see `tooltip` for details on how this affects the tooltip behavior).
`icon`	An icon. If provided without `show_label = TRUE`, only the icon will be visible.
`show_label`	Whether to show the label text. If `FALSE` (the default), only the icon is shown (if provided). If `TRUE`, the label text is shown alongside the icon. Note that `show_label` can be dynamically updated using `update_toolbar_input_button()`.
`tooltip`	Tooltip text to display when hovering over the input. Can be: • `TRUE` (default when `show_label = FALSE`) - shows a tooltip with the `label` text • `FALSE` (default when `show_label = TRUE`) - no tooltip • A character string - shows a tooltip with custom text

	Defaults to `!show_label`. When a tooltip is created, it will have an ID of `"{id}_tooltip"` which can be used to update the tooltip text dynamically via `update_tooltip()`.
`...`	Additional attributes to pass to the button.
`disabled`	If `TRUE`, the button will not be clickable. Use `update_toolbar_input_button()` to dynamically enable/disable the button.
`border`	Whether to show a border around the button.
`session`	A Shiny session object (the default should almost always be used).

Value

Returns a button suitable for use in a toolbar.

Functions

- `toolbar_input_button()`: Create a toolbar button.
- `update_toolbar_input_button()`: Update a toolbar button.

Updating toolbar buttons

You can dynamically update the appearance and enabled/disabled state of a toolbar button on the client side using `update_toolbar_input_button()`. This function works similarly to `shiny::updateActionButton()`.

Note that you cannot change the `tooltip` or `border` parameters after the button has been created, as these affect the button's structure and ARIA attributes. Please use `update_tooltip()` to update the text of the tooltip if one is present.

For example:

```
library(shiny)
library(bslib)

ui <- page_fluid(
  card(
    card_header(
      "Toolbar Demo",
      toolbar(
        align = "right",
        toolbar_input_button("btn", label = "Click me", icon = icon("play")),
        toolbar_input_button(
          "task",
          label = "Do a Task",
          icon = icon("play-circle"),
          show_label = TRUE
        )
      )
    ),
    card_body(
      verbatimTextOutput("count")
```

```
    )
  )
)

server <- function(input, output, session) {
  output$count <- renderPrint({
    input$btn
  })

  observeEvent(input$btn, {
    update_toolbar_input_button(
      "btn",
      label = "Clicked!",
      icon = icon("check")
    )
    # Update the tooltip text
    update_tooltip("btn_tooltip", "Button was clicked!")
  })

  # Handle task button - toggle between states
  observeEvent(input$task, {
    if (input$task %% 2 == 1) {
      # Show task is in progress
      update_toolbar_input_button(
        "task",
        label = "Task Running...",
        icon = icon("spinner")
      )
    } else {
      # Reset to original
      update_toolbar_input_button(
        "task",
        label = "Do a Task",
        icon = icon("play-circle"),
        session = session
      )
    }
  })
}

shinyApp(ui, server)
```

See Also

Other toolbar components: `toolbar()`, `toolbar_divider()`, `toolbar_input_select()`

Examples

```
# Basic toolbar button
```

```
toolbar(
  toolbar_input_button(id = "view", icon = icon("eye"), label = "View"),
  toolbar_input_button(id = "save", icon = icon("save"), label = "Save")
)
```

`toolbar_input_select` *Toolbar Input Select*

Description

Create a select input control that can be used to choose a single item from a list of values, suitable for use within a `toolbar()`.

Usage

```
toolbar_input_select(
  id,
  label,
  choices,
  ...,
  selected = NULL,
  icon = NULL,
  show_label = FALSE,
  tooltip = !show_label
)

update_toolbar_input_select(
  id,
  label = NULL,
  show_label = NULL,
  choices = NULL,
  selected = NULL,
  icon = NULL,
  session = get_current_session()
)
```

Arguments

`id`	The input ID.
`label`	The input label. By default, `label` is not shown but is used by `tooltip`. Set `show_label = TRUE` to show the label (see `tooltip` for details on how this affects the tooltip behavior).
`choices`	List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the

group labels (leveraging the <optgroup> HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature.

`...` Additional named arguments passed as attributes to the outer container div.

`selected` The initially selected value. If not provided on input creation, the first choice will be selected by default. If provided in `update_toolbar_input_select()` with a new set of `choices`, it will replace the currently selected value.

`icon` An icon. If provided without `show_label = TRUE`, only the icon will be visible.

`show_label` Whether to show the label text. If `FALSE` (the default), only the icon is shown (if provided). If `TRUE`, the label text is shown alongside the icon. Note that `show_label` can be dynamically updated using `update_toolbar_input_button()`.

`tooltip` Tooltip text to display when hovering over the input. Can be:

- `TRUE` (default when `show_label = FALSE`) - shows a tooltip with the `label` text
- `FALSE` (default when `show_label = TRUE`) - no tooltip
- A character string - shows a tooltip with custom text

Defaults to `!show_label`. When a tooltip is created, it will have an ID of `"{id}_tooltip"` which can be used to update the tooltip text dynamically via `update_tooltip()`.

`session` A Shiny session object (the default should almost always be used).

Details

When a tooltip is created for the select input, it will have an ID of `"{id}_tooltip"` which can be used to update the tooltip text dynamically via `update_tooltip()`.

Value

Returns a select input control suitable for use in a toolbar.

Functions

- `toolbar_input_select()`: Create a toolbar select input.
- `update_toolbar_input_select()`: Update a toolbar select input.

Updating toolbar select inputs

You can update the appearance and choices of a toolbar select input. This function works similarly to `shiny::updateSelectInput()`, but is specifically designed for `toolbar_input_select()`. It allows you to update the select's label, icon, choices, selected value, and label visibility from the server.

Note that you cannot enable or disable the `tooltip` parameter after the select has been created, only update the text of the tooltip. When a tooltip is created for the select input, it will have an ID of `"{id}_tooltip"` which can be used to update the tooltip text dynamically via `update_tooltip()`.

For example:

```
library(shiny)
library(bslib)

ui <- page_fluid(
  toolbar(
    align = "right",
    toolbar_input_select(
      "select",
      label = "Choose",
      icon = bsicons::bs_icon("filter"),
      choices = c("A", "B", "C")
    ),
    toolbar_input_button(
      "change_choices",
      label = "Change Choices",
      show_label = TRUE,
      icon = bsicons::bs_icon("arrow-repeat")
    )
  ),
  verbatimTextOutput("value")
)

server <- function(input, output, session) {
  output$value <- renderPrint({
    input$select
  })

  observeEvent(input$change_choices, {
    update_toolbar_input_select(
      "select",
      label = "Pick one",
      choices = c("hi", "hello", "hey"),
      selected = "hello"
    )
    # Update the tooltip text
    update_tooltip("select_tooltip", "Choose your NEW option")
  })
}

shinyApp(ui, server)
```

See Also

Other toolbar components: toolbar(), toolbar_divider(), toolbar_input_button()

Examples

```
toolbar(
  align = "right",
```

```
  toolbar_input_select(
    id = "select",
    label = "Choose option",
    choices = c("Option 1", "Option 2", "Option 3"),
    selected = "Option 2"
  )
)
```

`tooltip` *Add a tooltip to a UI element*

Description

Display additional information when focusing (or hovering over) a UI element.

Usage

```
tooltip(
  trigger,
  ...,
  id = NULL,
  placement = c("auto", "top", "right", "bottom", "left"),
  options = list()
)

toggle_tooltip(id, show = NULL, session = get_current_session())

update_tooltip(id, ..., session = get_current_session())
```

Arguments

`trigger`	A UI element (i.e., htmltools tag) to serve as the tooltip trigger. If `trigger` renders as multiple HTML elements (e.g., it's a `tagList()`), the last HTML element is used for the trigger. If the `trigger` should contain all of those elements, wrap the object in a `htmltools::div()` or `htmltools::span()`.
`...`	UI elements for the tooltip. Character strings are automatically escaped unless marked as `htmltools::HTML()`. Tooltip content should expand on, not contradict element labels.
`id`	a character string that matches an existing tooltip id.
`placement`	The placement of the tooltip relative to its trigger.
`options`	A list of additional options.
`show`	Whether to show (`TRUE`) or hide (`FALSE`) the tooltip. The default (`NULL`) will show if currently hidden and hide if currently shown. Note that a tooltip will not be shown if the trigger is not visible (e.g., it's hidden behind a tab).
`session`	A Shiny session object (the default should almost always be used).

Functions

- `tooltip()`: Add a tooltip to a UI element
- `toggle_tooltip()`: Programmatically show/hide a tooltip.
- `update_tooltip()`: Update the contents of a tooltip.

Theming/Styling

Like other bslib components, tooltips can be themed by supplying relevant theming variables to `bs_theme()`, which effects styling of every tooltip on the page. To style a *specific* tooltip differently from other tooltip, utilize the `customClass` option:

```
tooltip(
  "Trigger", "Tooltip message",
  options = list(customClass = "my-tip")
)
```

And then add relevant rules to `bs_theme()` via `bs_add_rules()`:

```
bs_theme() |> bs_add_rules(".my-tip { max-width: none; }")
```

Accessibility of Tooltip Triggers

Because the user needs to interact with the `trigger` element to see the tooltip, it's best practice to use an element that is typically accessible via keyboard interactions, like a button or a link. If you use a non-interactive element, like a `<span>` or text, bslib will automatically add the `tabindex="0"` attribute to the trigger element to make sure that users can reach the element with the keyboard. This means that in most cases you can use any element you want as the trigger.

One place where it's important to consider the accessibility of the trigger is when using an icon without any accompanying text. In these cases, many R packages that provide icons will create an icon element with the assumption that the icon is decorative, which will make it inaccessible to users of assistive technologies.

When using an icon as the primary trigger, ensure that the icon does not have `aria-hidden="true"` or `role="presentation"` attributes. Icon packages typically provide a way to specify a title for the icon, as well as a way to specify that the icon is not decorative. The title should be a short description of the purpose of the trigger, rather than a description of the icon itself.

- If you're using `bsicons::bs_icon()`, provide a `title`.
- If you're using `fontawesome::fa()`, set `a11y = "sem"` and provide a `title`.

For example:

```
tooltip(
  bsicons::bs_icon("info-circle", title = "About tooltips"),
  "Text shown in the tooltip."
)

tooltip(
  fontawesome::fa("info-circle", a11y = "sem", title = "About tooltips"),
  "Text shown in the tooltip."
)
```

References

Tooltips are based on Bootstrap's Tooltip component. See the bslib website for an interactive introduction to tooltips and popovers.

See Also

`popover()` provides a an alternative and more persistent container for additional elements, typically revealed by clicking on a target element.

Other Components: `accordion()`, `card()`, `popover()`, `value_box()`

Examples

```
tooltip(
  shiny::actionButton("btn", "A button"),
  "A message"
)

card(
  card_header(
    tooltip(
      span("Card title ", bsicons::bs_icon("question-circle-fill")),
      "Additional info",
      placement = "right"
    )
  ),
  "Card body content..."
)
```

`value_box` *Value box*

Description

An opinionated (`card()`-powered) box, designed for displaying a `value` and `title`. Optionally, a `showcase` can provide for context for what the `value` represents (for example, it could hold a `bsicons::bs_icon()`, or even a `shiny::plotOutput()`). Find examples and template code you can use to create engaging value boxes on the bslib website.

Usage

```
value_box(
  title,
  value,
  ...,
  showcase = NULL,
  showcase_layout = c("left center", "top right", "bottom"),
```

```
  full_screen = FALSE,
  theme = NULL,
  height = NULL,
  max_height = NULL,
  min_height = NULL,
  fill = TRUE,
  class = NULL,
  id = NULL,
  theme_color = deprecated()
)

value_box_theme(name = NULL, bg = NULL, fg = NULL)

showcase_left_center(
  width = 0.3,
  width_full_screen = "1fr",
  max_height = "100px",
  max_height_full_screen = 0.67
)

showcase_top_right(
  width = 0.4,
  width_full_screen = "1fr",
  max_height = "75px",
  max_height_full_screen = 0.67
)

showcase_bottom(
  width = "100%",
  width_full_screen = NULL,
  height = "auto",
  height_full_screen = "2fr",
  max_height = "100px",
  max_height_full_screen = NULL
)
```

Arguments

title, value	A string, number, or `htmltools::tag()` child to display as the title or value of the value box. The `title` appears above the `value`.
...	Unnamed arguments may be any `htmltools::tag()` children to display below `value`. Named arguments become attributes on the containing element.
showcase	A `htmltools::tag()` child to showcase (e.g., a `bsicons::bs_icon()`, a `plotly::plotlyOutput()`, etc).
showcase_layout	One of `"left center"` (default), `"top right"` or `"bottom"`. Alternatively, you can customize the showcase layout options with the `showcase_left_center()`,

showcase_top_right(), or showcase_bottom() functions. Use the options functions when you want to control the height or width of the showcase area.

full_screen
: If TRUE, an icon will appear when hovering over the card body. Clicking the icon expands the card to fit viewport size.

theme
: The name of a theme for the value box, or a theme constructed with value_box_theme(). The theme names provide a convenient way to use your app's Bootstrap theme colors as the foreground or background colors of the value box. See below for more details on the provided themes. For more control, you can create your own theme with value_box_theme() where you can pass foreground and background colors directly. See the **Themes** section for more details.

max_height
: The maximum height of the value_box() or the showcase area when used in a showcase_layout_*() function. Can be any valid CSS unit (e.g., max_height="200px").

min_height
: The minimum height of the values box. Can be any valid CSS unit (e.g., min_height="200px").

fill
: Whether to allow the value box to grow/shrink to fit a fillable container with an opinionated height (e.g., page_fillable()).

class
: Utility classes for customizing the appearance of the summary card. Use bg-* and text-* classes (e.g, "bg-danger" and "text-light") to customize the background/foreground colors.

id
: Provide a unique identifier for the card() or value_box() to report its full screen state to Shiny. For example, using id = "my_card", you can observe the card's full screen state with input$my_card_full_screen.

theme_color
: **[Deprecated]** Use theme instead.

name
: The name of the theme, e.g. "primary", "danger", "purple".

bg, fg
: The background and foreground colors for the theme. If only bg is provided, then the foreground color is automatically chosen from $black or $white to provide the best contrast with the background color.

width, width_full_screen, height, height_full_screen
: one of the following:

  - A proportion (i.e., a number between 0 and 1) of available width or height to allocate to the showcase.
  - A valid CSS unit defining the width or height of the showcase column, or a valid value accepted by the grid-template-columns (width) or grid-template-rows (height) CSS property to define the width or height of the showcase column or row. Accepted values in the second category are "auto", "min-content", "max-content", a fractional unit (e.g. 2fr), or a minmax() function (e.g., minmax(100px, 1fr)).

max_height_full_screen
: A proportion (i.e., a number between 0 and 1) or any valid CSS unit defining the showcase max_height in a full screen card.

Build-a-Box App

Explore all of the value_box() options and layouts interactively with the Build-a-Box app, available online thanks to shinyapps.io. Or, you can run the app locally with:

```
# shiny >= 1.8.1
shiny::runExample("build-a-box", package = "bslib")

# shiny < 1.8.1
shiny::runApp(system.file("examples-shiny", "build-a-box", package = "bslib"))
```

Themes

The appearance of a `value_box()` can be controlled via the `theme` argument in one of two ways:

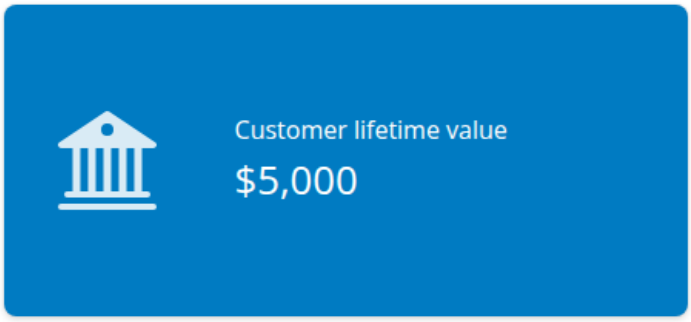
1. a character value describing the theme, such as `theme = "primary"` or `theme = "blue"`; or
2. `theme = value_box_theme()` to create a custom theme.

We recommend using named themes for most value boxes (the first approach), because these themes will automatically match your Bootstrap theme.

Named themes:

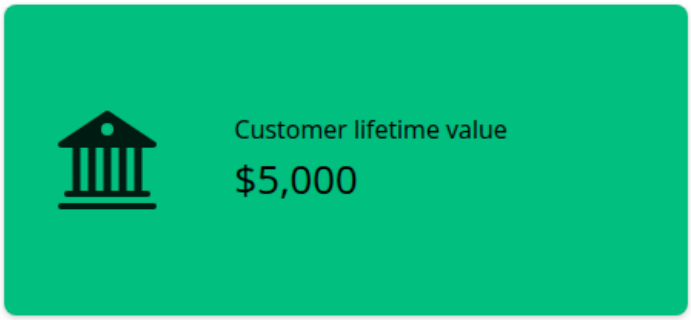
Bootstrap provides a list of theme colors, with semantic names like `"primary"`, `"secondary"`, `"success"`, `"danger"`, etc. You can set `theme` to one of these names to use the corresponding theme color as the background color of your value box.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "primary"
)
```



Bootstrap's theme colors are drawn from a second color list that includes variations on several main colors, named literally. These colors include `"blue"`, `"purple"`, `"pink"`, `"red"`, `"orange"`, `"yellow"`, `"green"`, `"teal"`, and `"cyan"`.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "teal"
)
```

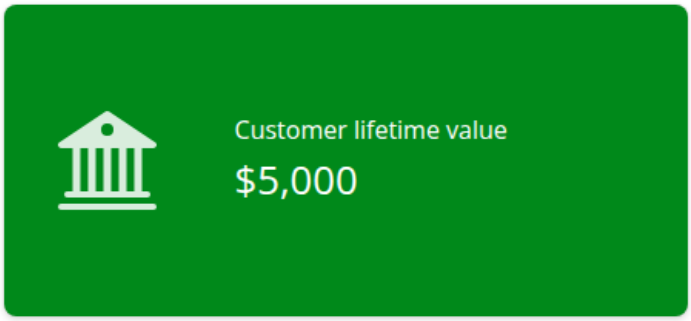


Background colors:

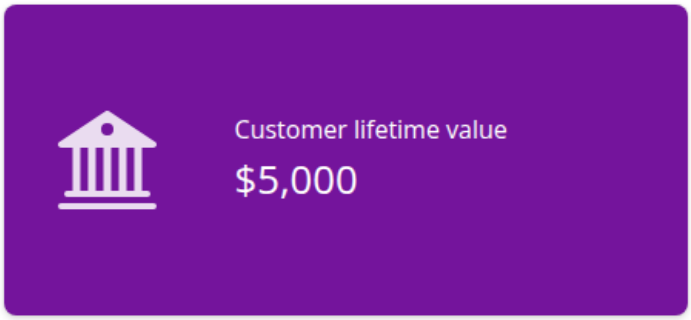
If the theme or color name is provided without any prefix, the color will be used for the background of the value box. You can also explicitly prefix the theme or color name with `bg-` to indicate that it should apply to the value box background. When the theme sets the background color, either black or white is chosen automatically for the text color using Bootstrap's color contrast algorithm.

As before, you can reference semantic theme color names or literal color names.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "bg-success"
)
```



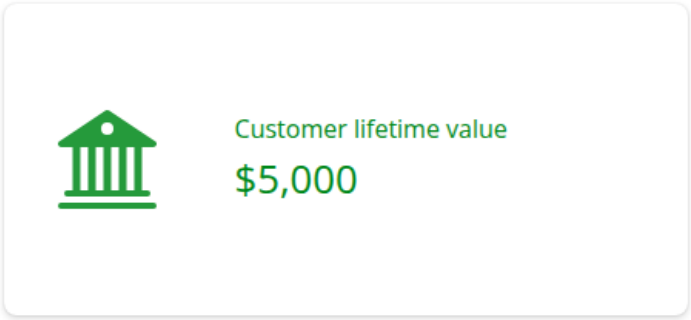

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "bg-purple"
)
```



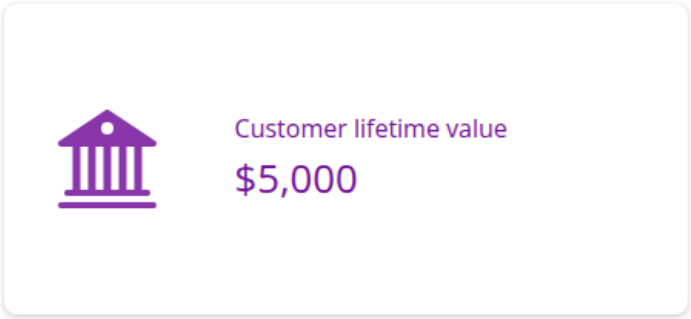
Foreground colors:

To set only the foreground colors of the value box, you can prefix the theme or color name with `text-`. This changes the text color without affecting the background color.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "text-success"
)
```

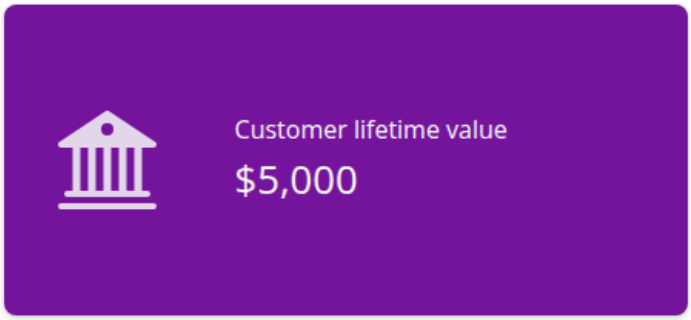



```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "text-purple"
)
```



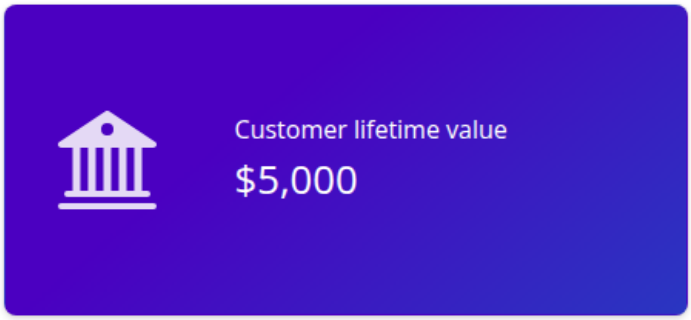
Occasionally you may want to adjust use both background and foreground themes on your value box. To achieve this, set `theme` to one of the theme names and use `class` for the complementary style. The example below uses `theme = "purple"` (which could also be `"bg-purple"`) for a purple background, and `class = "text-light"` for light-colored text.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "purple",
  class = "text-light"
)
```



Gradient backgrounds:
For a vibrant and attention-grabbing effect, bslib provides an array of gradient background options. Provide `theme` with a theme name in the form `bg-gradient-{from}-{to}`, where `{from}` and `{to}` are named main colors, e.g. `bg-gradient-indigo-blue`.

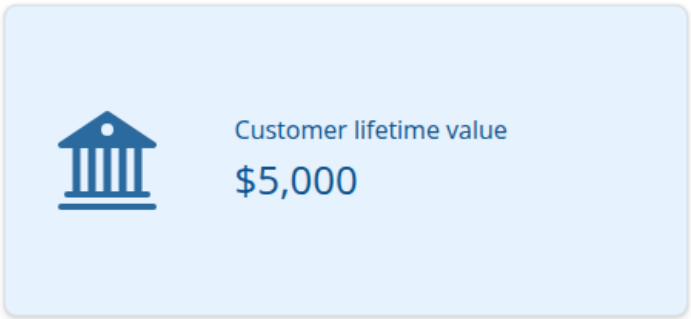
```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = "bg-gradient-indigo-blue"
)
```



Custom colors:

Finally, for complete customization, you can use `value_box_theme()` to create a custom theme. This function takes arguments `bg` and `fg` to set the background and foreground colors, respectively. Like with the `bg-` theme names, if only `bg` is provided, `value_box_theme()` will choose an appropriate light or dark color for the text color.

```
value_box(
  title = "Customer lifetime value",
  value = "$5,000",
  showcase = bsicons::bs_icon("bank2"),
  theme = value_box_theme(bg = "#e6f2fd", fg = "#0B538E"),
  class = "border"
)
```



Note that `value_box_theme()` optionally takes a `theme` name, which can be helpful if you want to use a named theme and modify the default `bg` or `fg` colors of that theme.

```
value_box_theme(name = "orange", bg = "#FFFFFF")
value_box_theme(name = "text-danger", fg = "#FFB6C1")
```

Also note that `bg`/`fg` *must* be CSS colors, not Bootstrap theme or color names. This means that `theme = "purple"` will use your Bootstrap theme's purple color, and `bg = "purple"` will use the CSS color for *purple*, i.e. `"#800080"`.

Showcase Layouts

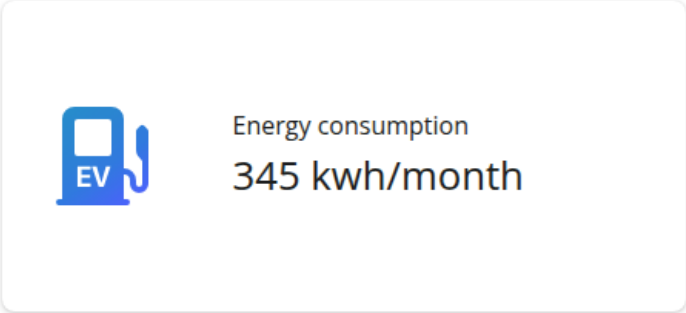
Use the `showcase` argument to add a plot or icon to your `value_box()`. There are three layouts available: `"left center"`, `"top right"`, and `"bottom"`. Set `showcase` to the name of the layout you'd like, or use the `showcase_left_center()`, `showcase_top_right()`, or `showcase_bottom()` helper functions to customize the showcase area's size.

If you're using a plot as your showcase, you may also want to set `fullscreen = TRUE` so that your users can expand the value box into a full screen card. See the value box article for more details.

Left-center showcase:

The `"left center"` showcase layout is the default, and is perfect for an icon or a small plot. This layout works best for short value boxes.

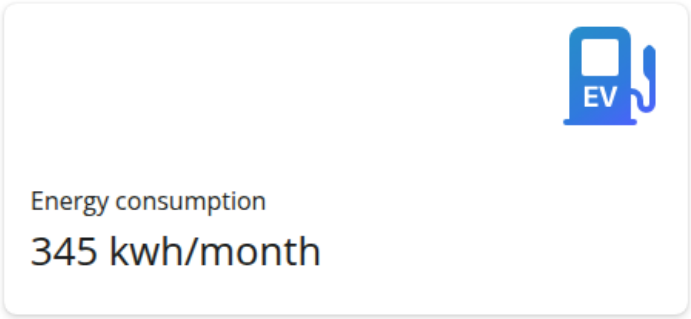
```
value_box(
  title = "Energy consumption",
  value = "345 kwh/month",
  showcase = bsicons::bs_icon("ev-station-fill")
)
```



Top-right showcase:

The `"top right"` showcase layout places the icon or plot in the upper right corner of the value box. This layout works best for medium-height to square value boxes.

```
value_box(
  title = "Energy consumption",
  value = "345 kwh/month",
  showcase = bsicons::bs_icon("ev-station-fill"),
  showcase_layout = "top right"
)
```



Bottom showcase:

Finally, the `"bottom"` showcase layout is perfect for full-bleed plots. This layout places the plot below the title and value, with the plot taking up the full width of the bottom half.

Try this layout with sparkline-style plots. These can be a little tricky to set up, so be sure to check out the Expandable sparklines section of the value boxes article on the bslib website. In this example, we've created a sparkline plot using base R graphics, which isn't generally recommended. View the bslib documentation online to see the source of `sparkline_plot()`.

```
value_box(
  title = "Energy consumption",
  value = "345 kwh/month",
  showcase = sparkline_plot(),
  showcase_layout = "bottom"
)
```

References

Value boxes are featured on the bslib website in a few articles:

- Value boxes
- Build-a-Box App
- Get Started: Dashboards

See Also

Value boxes are a specialized form of a `card()` component.

`layout_columns()` and `layout_column_wrap()` help position multiple value boxes into columns and rows.

Other Components: `accordion()`, `card()`, `popover()`, `tooltip()`

Examples

```
library(htmltools)

value_box(
  "KPI Title",
  h1(HTML("$1 <i>Billion</i> Dollars")),
  span(
    bsicons::bs_icon("arrow-up"),
    " 30% VS PREVIOUS 30 DAYS"
  ),
  showcase = bsicons::bs_icon("piggy-bank"),
  theme = "success"
)
```

---

versions *Available Bootstrap versions*

Description

Available Bootstrap versions

Usage

```
versions()

version_default()
```

Value

Returns a list of the Bootstrap versions available.

See Also

Other Bootstrap theme utility functions: `bootswatch_themes()`, `bs_get_variables()`, `builtin_themes()`, `theme_bootswatch()`, `theme_version()`

Index