Package: shiny.fluent (via r-universe)

June 1, 2026

Title Microsoft Fluent UI for Shiny Apps

Version 0.4.0

Description A rich set of UI components for building Shiny applications, including inputs, containers, overlays, menus, and various utilities. All components from Fluent UI (the underlying JavaScript library) are available and have usage examples in R.

URL https://appsilon.github.io/shiny.fluent/, https://github.com/appsilon/shiny.fluent

License LGPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.1

Depends R (>= 2.10)

Imports htmltools, jsonlite, purrr, shiny, shiny.react (>= 0.4.0)

Suggests chromote, covr, dplyr, DT, ggplot2, glue, imola, knitr, leaflet, mockery, plotly, rcmdcheck, RColorBrewer, rmarkdown, sass, shiny.i18n (>= 0.3.0), shiny.router (>= 0.3.1), shinyjs, shinytest2, sortable, stringi, testthat (>= 3.0.0), tibble, withr

Config/testthat/edition 3

NeedsCompilation no

Author Jakub Sobolewski [aut, cre], Kamil Żyła [aut], Marek Rogala [aut], Appsilon Sp. z o.o. [cph]

Maintainer Jakub Sobolewski <opensource+jakub.sobolewski@appsilon.com>

Config/pak/sysreqs cmake make libicu-dev libuv1-dev zlib1g-dev

Repository https://cranhaven.r-universe.dev

Date/Publication 2026-06-01 04:02:00 UTC

RemoteUrl https://github.com/cranhaven/cranhaven.r-universe.dev

RemoteRef package/shiny.fluent

RemoteSha 06a82e74039f47a4d3e1d05b0fab9b680d8e6bbb

RemoteSubdir shiny.fluent

Contents

`ActionButton` *Button*

Description

Buttons give people a way to trigger an action. They're typically found in forms, dialog panels, and dialogs. Some buttons are specialized for particular tasks, such as navigation, repeated actions, or presenting menus.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ActionButton(...)

CommandBarButton(...)

CommandButton(...)
```

```
CompoundButton(...)

DefaultButton(...)

IconButton(...)

PrimaryButton(...)

ActionButton.shinyInput(inputId, ...)

updateActionButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)

CommandBarButton.shinyInput(inputId, ...)

updateCommandBarButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)

CommandButton.shinyInput(inputId, ...)

updateCommandButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)

CompoundButton.shinyInput(inputId, ...)

updateCompoundButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)

DefaultButton.shinyInput(inputId, ...)

updateDefaultButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

```
IconButton.shinyInput(inputId, ...)

updateIconButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)

PrimaryButton.shinyInput(inputId, ...)

updatePrimaryButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **baseClassName** `string`
- **variantClassName** `string`
- **allowDisabledFocus** `boolean`
  Whether the button can have focus in disabled mode
- **ariaDescription** `string`
  Detailed description of the button for the benefit of screen readers.

Besides the compound button, other button types will need more information provided to screen reader.

- **ariaHidden** `boolean`
  If provided and is true it adds an 'aria-hidden' attribute instructing screen readers to ignore the element.
- **ariaLabel** `string`
  The aria label of the button for the benefit of screen readers.
- **buttonType** `ButtonType`
  Deprecated at v1.2.3, to be removed at \>= v2.0.0. Use specific button component instead.
- **checked** `boolean`
  Whether the button is checked

- **className** `string`
  If provided, additional class name to provide on the root element.
- **componentRef** `IRefObject<IButton>`
  Optional callback to access the IButton interface. Use this instead of ref for accessing the public methods and properties of the component.
- **data** `any`
  Any custom data the developer wishes to associate with the menu item.
- **defaultRender** `any`
  yet unknown docs
- **description** `IStyle`
  Style for the description text if applicable (for compound buttons.) Deprecated, use `secondaryText` instead.
- **disabled** `boolean`
  Whether the button is disabled
- **getClassNames** `(theme: ITheme, className: string, variantClassName: string, iconClassName: string`
  Method to provide the classnames to style a button. The default value for this prop is the getClassnames func defined in BaseButton.classnames.
- **getSplitButtonClassNames** `(disabled: boolean, expanded: boolean, checked: boolean, allowDisabledFoc`
  Method to provide the classnames to style a button. The default value for this prop is the getClassnames func defined in BaseButton.classnames.
- **href** `string`
  If provided, this component will be rendered as an anchor.
- **iconProps** `IIconProps`
  The props for the icon shown in the button.
- **keytipProps** `IKeytipProps`
  Optional keytip for this button
- **menuAs** `IComponentAs<IContextualMenuProps>`
  Render a custom menu in place of the normal one.
- **menuIconProps** `IIconProps`
  The props for the icon shown when providing a menu dropdown.
- **menuProps** `IContextualMenuProps`
  Props for button menu. Providing this will default to showing the menu icon. See menuIconProps for overriding how the default icon looks. Providing this in addition of onClick and setting the split property to true will render a SplitButton.
- **menuTriggerKeyCode** `KeyCodes | null`
  Provides a custom KeyCode that can be used to open the button menu. The default KeyCode is the down arrow. A value of null can be provided to disable the key codes for opening the button menu.
- **onAfterMenuDismiss** `() => void`
  Callback that runs after Button's contextualmenu was closed (removed from the DOM)
- **onMenuClick** `(ev?: React.MouseEvent<HTMLElement> | React.KeyboardEvent<HTMLElement>, button?: IBu`
  Optional callback when menu is clicked.
- **onRenderAriaDescription** `IRenderFunction<IButtonProps>`
  Custom render function for the aria description element.

- **onRenderChildren** `IRenderFunction<IButtonProps>`
  Custom render function for rendering the button children.
- **onRenderDescription** `IRenderFunction<IButtonProps>`
  Custom render function for the desciption text.
- **onRenderIcon** `IRenderFunction<IButtonProps>`
  Custom render function for the icon
- **onRenderMenu** `IRenderFunction<IContextualMenuProps>`
  Deprecated at v6.3.2, to be removed at \>= v7.0.0. Use `menuAs` instead.
- **onRenderMenuIcon** `IRenderFunction<IButtonProps>`
  Custom render function for button menu icon
- **onRenderText** `IRenderFunction<IButtonProps>`
  Custom render function for the label text.
- **persistMenu** `boolean`
  Menu will not be created or destroyed when opened or closed, instead it will be hidden. This will improve perf of the menu opening but could potentially impact overall perf by having more elements in the dom. Should only be used when perf is important. Note: This may increase the amount of time it takes for the button itself to mount.
- **primary** `boolean`
  Changes the visual presentation of the button to be emphasized (if defined)
- **primaryActionButtonProps** `IButtonProps`
  Optional props to be applied only to the primary action button of SplitButton and not to the overall SplitButton container
- **primaryDisabled** `boolean`
  If set to true and if this is a splitButton (split == true) then the primary action of a split button is disabled.
- **renderPersistedMenuHiddenOnMount** `boolean`
  If true, the persisted menu is rendered hidden when the button initially mounts. Non-persisted menus will not be in the component tree unless they are being shown

Note: This increases the time the button will take to mount, but can improve perceived menu open perf. when the user opens the menu.

- **rootProps** `React.ButtonHTMLAttributes<HTMLButtonElement> | React.AnchorHTMLAttributes<HTMLAnchorE`
  Deprecated at v0.56.2, to be removed at \>= v1.0.0. Just pass in button props instead. they will be mixed into the button/anchor element rendered by the component.
- **secondaryText** `string`
  Description of the action this button takes. Only used for compound buttons
- **split** `boolean`
  If set to true, and if menuProps and onClick are provided, the button will render as a SplitButton.
- **splitButtonAriaLabel** `string`
  Accessible label for the dropdown chevron button if this button is split.
- **splitButtonMenuProps** `IButtonProps`
  Experimental prop that get passed into the menuButton that's rendered as part of split button. Anything passed in will likely need to have accompanying style changes.

- **styles** `IButtonStyles`
  Custom styling for individual elements within the button DOM.
- **text** `string`
  Text to render button label. If text is supplied, it will override any string in button children. Other children components will be passed through after the text.
- **theme** `ITheme`
  Theme provided by HOC.
- **toggle** `boolean`
  Whether button is a toggle button with distinct on and off states. This should be true for buttons that permanently change state when a press event finishes, such as a volume mute button.
- **toggled** `boolean`
  Any custom data the developer wishes to associate with the menu item. Deprecated, use `checked` if setting state.
- **uniqueId** `string | number`
  Unique id to identify the item. Typically a duplicate of key value.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- For dialog boxes and panels, where people are moving through a sequence of screens, right-align buttons with the container.
- For single-page forms and focused tasks, left-align buttons with the container.
- Always place the primary button on the left, the secondary button just to the right of it.
- Show only one primary button that inherits theme color at rest state. If there are more than two buttons with equal priority, all buttons should have neutral backgrounds.
- Don't use a button to navigate to another place; use a link instead. The exception is in a wizard where "Back" and "Next" buttons may be used.
- Don't place the default focus on a button that destroys data. Instead, place the default focus on the button that performs the "safe act" and retains the content (such as "Save") or cancels the action (such as "Cancel").

Content:

- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.
- Make sure it's clear what will happen when people interact with the button. Be concise; usually a single verb is best. Include a noun if there is any room for interpretation about what the verb means. For example, "Delete folder" or "Create account".

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

tokens <- list(childrenGap = 20)

ui <- function(id) {
  ns <- NS(id)
  tags$div(
    Stack(
      DefaultButton.shinyInput(
        ns("button1"),
        text = "Default Button",
        styles = list("background: green")
      ),
      PrimaryButton.shinyInput(
        ns("button2"),
        text = "Primary Button"
      ),
      CompoundButton.shinyInput(
        ns("button3"),
        secondaryText = "Compound Button has additional text",
        text = "Compound Button"
      ),
      ActionButton.shinyInput(
        ns("button4"),
        iconProps = list("iconName" = "AddFriend"),
        text = "Action Button"
      ),
      horizontal = TRUE,
      tokens = tokens
    ),
    textOutput(ns("text"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    clicks <- reactiveVal(0)
    addClick <- function() { clicks(isolate(clicks() + 1)) }
    observeEvent(input$button0, addClick())
    observeEvent(input$button1, addClick())
    observeEvent(input$button2, addClick())
    observeEvent(input$button3, addClick())
    observeEvent(input$button4, addClick())
    output$text <- renderText({
      paste0("Clicks:", clicks())
    })
  })
}
```

```
if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Split button with menu
menuProps <- list(
  items = list(
    list(
      key = "emailMessage",
      text = "Email message",
      onClick = JS("() => alert('Email message clicked')"),
      iconProps = list(
        iconName = "Mail"
      )
    ),
    list(
      key = "calendarEvent",
      text = "Calendar event",
      onClick = JS("() => alert('Calendar event clicked')"),
      iconProps = list(
        iconName = "Calendar"
      )
    )
  )
)

ui <- function(id) {
  ns <- NS(id)
  fluentPage(
    Stack(
      horizontal = TRUE,
      wrap = TRUE,
      tokens = list(
        childrenGap = 40
      ),
      DefaultButton.shinyInput(
        inputId = ns("button_1"),
        text = "Standard",
        primary = FALSE,
        split = TRUE,
        splitButtonAriaLabel = "See 2 options",
        `aria-roledescription` = "split button",
        menuProps = menuProps,
        disabled = FALSE,
        checked = FALSE
      ),
      DefaultButton.shinyInput(
        inputId = ns("button_2"),
        text = "Primary",
```

```
        primary = TRUE,
        split = TRUE,
        splitButtonAriaLabel = "See 2 options",
        `aria-roledescription` = "split button",
        menuProps = menuProps,
        disabled = FALSE,
        checked = FALSE
      ),
      DefaultButton.shinyInput(
        inputId = ns("button_3"),
        text = "Main action disabled",
        primaryDisabled = NA,
        split = TRUE,
        splitButtonAriaLabel = "See 2 options",
        `aria-roledescription` = "split button",
        menuProps = menuProps,
        checked = FALSE
      ),
      DefaultButton.shinyInput(
        inputId = ns("button_4"),
        text = "Disabled",
        disabled = TRUE,
        split = TRUE,
        splitButtonAriaLabel = "See 2 options",
        `aria-roledescription` = "split button",
        menuProps = menuProps,
        checked = FALSE
      )
    ),
    uiOutput(ns("text"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$text <- renderUI({
      lapply(seq_len(4), function(i) {
        paste0("button_", i, ": ", input[[paste0("button_", i)]])
      })
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 3
library(shiny)
library(shiny.fluent)
library(shinyjs)

# This example app shows how to use a Fluent UI Button to trigger a file upload.
```

```
# File upload is not natively supported by shiny.fluent so shinyjs is used
# to trigger the file upload input.
ui <- function(id) {
  ns <- NS(id)
  fluentPage(
    useShinyjs(),
    Stack(
      tokens = list(
        childrenGap = 10L
      ),
      horizontal = TRUE,
      DefaultButton.shinyInput(
        inputId = ns("uploadFileButton"),
        text = "Upload File",
        iconProps = list(iconName = "Upload")
      ),
      div(
        style = "
          visibility: hidden;
          height: 0;
          width: 0;
        ",
        fileInput(
          inputId = ns("uploadFile"),
          label = NULL
        )
      )
    ),
    textOutput(ns("file_path"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    observeEvent(input$uploadFileButton, {
      click("uploadFile")
    })

    output$file_path <- renderText({
      input$uploadFile$name
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 4
library(shiny)
library(shiny.fluent)
library(shinyjs)
```

```
# This example app shows how to use a Fluent UI Button to trigger a file download.
# File download is not natively supported by shiny.fluent so shinyjs is used
# to trigger the file download.
ui <- function(id) {
  ns <- NS(id)
  fluentPage(
    useShinyjs(),
    DefaultButton.shinyInput(
      inputId = ns("downloadButton"),
      text = "Download",
      iconProps = list(iconName = "Download")
    ),
    div(
      style = "visibility: hidden;",
      downloadButton(ns("download"), label = "")
    )
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    observeEvent(input$downloadButton, {
      click("download")
    })

    output$download <- downloadHandler(
      filename = function() {
        paste("data-", Sys.Date(), ".csv", sep="")
      },
      content = function(file) {
        write.csv(iris, file)
      }
    )
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`ActivityItem` *ActivityItem*

Description

An activity item (`ActivityItem`) represents a person's actions, such as making a comment, mentioning someone with an @mention, editing a document, or moving a file.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ActivityItem(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **activityDescription** `React.ReactNode[] | React.ReactNode`
  An element describing the activity that took place. If no activityDescription, activityDescriptionText, or onRenderActivityDescription are included, no description of the activity is shown.
- **activityDescriptionText** `string`
  Text describing the activity that occurred and naming the people involved in it. Deprecated, use `activityDescription` instead.
- **activityIcon** `React.ReactNode`
  An element containing an icon shown next to the activity item.
- **activityPersonas** `Array<IPersonaSharedProps>`
  If activityIcon is not set, then the persona props in this array will be used as the icon for this activity item.
- **animateBeaconSignal** `boolean`
  Enables/Disables the beacon that radiates from the center of the center of the activity icon. Signals an activity has started.
- **beaconColorOne** `string`
  Beacon color one
- **beaconColorTwo** `string`
  Beacon color two
- **comments** `React.ReactNode[] | React.ReactNode`
  An element containing the text of comments or \@mention messages. If no comments, commentText, or onRenderComments are included, no comments are shown.
- **commentText** `string`
  Text of comments or \@mention messages. Deprecated, use `comments` instead.
- **isCompact** `boolean`
  Indicated if the compact styling should be used.
- **onRenderActivityDescription** `IRenderFunction<IActivityItemProps>`
  A renderer for the description of the current activity.
- **onRenderComments** `IRenderFunction<IActivityItemProps>`
  A renderer that adds the text of a comment below the activity description.
- **onRenderIcon** `IRenderFunction<IActivityItemProps>`
  A renderer to create the icon next to the activity item.
- **onRenderTimeStamp** `IRenderFunction<IActivityItemProps>`
  A renderer adds a time stamp. If not included, timeStamp is shown as plain text below the activity.

- **styles** `IActivityItemStyles`
  Optional styling for the elements within the Activity Item.
- **timeStamp** `string | React.ReactNode[] | React.ReactNode`
  Element shown as a timestamp on this activity. If not included, no timestamp is shown.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use a list of multiple activity items to indicate a history of events relating to a single file, folder, person, or other entity. Alternatively, use a single activity item to indicate the most recent event on an entity.
- Group multiple similar events occurring near the same time into a single activity item.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ActivityItem(
    activityDescription = tagList(
      Link(key = 1, "Philippe Lampros"),
      tags$span(key = 2, " commented")
    ),
    activityIcon = Icon(iconName = "Message"),
    comments = tagList(
      tags$span(key = 1, "Hello! I am making a comment.")
    ),
    timeStamp = "Just now"
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

Announced *Announced*

Description

The Announced component aims to fill several of the accessibility gaps that exist in various web application experiences. It provides text for the screen reader in certain scenarios that are lacking comprehensive updates, particularly those showing the completion status or progress of operation(s).

Some real-world applications of the component include copying, uploading, or moving many files; deleting or renaming a single file; "lazy loading" of page sections that do not appear all at once; and appearance of search results.

The Announced component currently has the following documented use cases:

1. Quick Actions: Operations such as editing text or deletion that are short enough that they do not require a status during progress.
2. Search Results: Appearance of search results such as in contact fields or search boxes.
3. Lazy Loading: "Lazy loading" of page sections that do not appear all at once.
4. Bulk Operations: Operations that require multiple sub operations, such as the moving of several files.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Announced(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **"aria-live"** `'off' | 'polite' | 'assertive'`
  Priority with which the screen reader should treat updates to this region @default 'polite'
- **as** `React.ElementType`
  Optionally render the root of this component as another component type or primitive. The custom type **must** preserve any children or native props passed in. @default 'div'
- **message** `string`
  The status message provided as screen reader output
- **styles** `IStyleFunctionOrObject<{}, IAnnouncedStyles>`
  Call to provide customized styling that will layer on top of the variant rules.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  Announced(message = "Screen reader message")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`BasePickerListBelow` *Pickers*

Description

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Pickers are used to select one or more items, such as tags or files, from a large list.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
BasePickerListBelow(...)

TagPicker(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  ClassName for the picker.
- **componentRef** `IRefObject<IBasePicker<T>>`
  Optional callback to access the IBasePicker interface. Use this instead of ref for accessing the public methods and properties of the component.
- **createGenericItem** `(input: string, ValidationState: ValidationState) => ISuggestionModel<T> | T`
  Function that specifies how arbitrary text entered into the well is handled.
- **defaultSelectedItems** `T[]`
  Initial items that have already been selected and should appear in the people picker.
- **disabled** `boolean`
  Flag for disabling the picker.
- **enableSelectedSuggestionAlert** `boolean`
  Adds an additional alert for the currently selected suggestion. This prop should be set to true for IE11 and below, as it enables proper screen reader behavior for each suggestion (since aria-activedescendant does not work with IE11). It should not be set for modern browsers (Edge, Chrome).
- **getTextFromItem** `(item: T, currentValue?: string) => string`
  A callback to get text from an item. Used to autofill text in the pickers.
- **inputProps** `IInputProps`
  AutoFill input native props
- **itemLimit** `number`
  Restrict the amount of selectable items.
- **onBlur** `React.FocusEventHandler<HTMLInputElement | Autofill>`
  A callback for when the user moves the focus away from the picker
- **onChange** `(items?: T[]) => void`
  A callback for when the selected list of items changes.
- **onDismiss** `(ev?: any, selectedItem?: T) => boolean | void`
  A callback to override the default behavior of adding the selected suggestion on dismiss. If it returns true or nothing, the selected item will be added on dismiss. If false, the selected item will not be added on dismiss.
- **onEmptyInputFocus** `(selectedItems?: T[]) => T[] | PromiseLike<T[]>`
  A callback for what should happen when a user clicks within the input area.
- **onEmptyResolveSuggestions** `(selectedItems?: T[]) => T[] | PromiseLike<T[]>`
  A callback for what should happen when suggestions are shown without input provided. Returns the already selected items so the resolver can filter them out. If used in conjunction with resolveDelay this will only kick off after the delay throttle.
- **onFocus** `React.FocusEventHandler<HTMLInputElement | Autofill>`
  A callback for when the user put focus on the picker
- **onGetMoreResults** `(filter: string, selectedItems?: T[]) => T[] | PromiseLike<T[]>`
  A callback that gets the rest of the results when a user clicks get more results.
- **onInputChange** `(input: string) => string`
  A callback used to modify the input string.

- **onItemSelected** `(selectedItem?: T) => T | PromiseLike<T> | null`
  A callback to process a selection after the user selects something from the picker. If the callback returns null, the item will not be added to the picker.
- **onRemoveSuggestion** `(item: T) => void`
  A callback for when an item is removed from the suggestion list
- **onRenderItem** `(props: IPickerItemProps<T>) => JSX.Element`
  Function that specifies how the selected item will appear.
- **onRenderSuggestionsItem** `(props: T, itemProps: ISuggestionItemProps<T>) => JSX.Element`
  Function that specifies how an individual suggestion item will appear.
- **onResolveSuggestions** `(filter: string, selectedItems?: T[]) => T[] | PromiseLike<T[]>`
  A callback for what should happen when a person types text into the input. Returns the already selected items so the resolver can filter them out. If used in conjunction with resolveDelay this will only kick off after the delay throttle.
- **onValidateInput** `(input: string) => ValidationState`
  A function used to validate if raw text entered into the well can be added into the selected items list
- **pickerCalloutProps** `ICalloutProps`
  The properties that will get passed to the Callout component.
- **pickerSuggestionsProps** `IBasePickerSuggestionsProps`
  The properties that will get passed to the Suggestions component.
- **removeButtonAriaLabel** `string`
  Aria label for the "X" button in the selected item component.
- **resolveDelay** `number`
  The delay time in ms before resolving suggestions, which is kicked off when input has been changed. e.g. If a second input change happens within the resolveDelay time, the timer will start over. Only until after the timer completes will onResolveSuggestions be called.
- **searchingText** `((props: { input: string; }) => string) | string`
  The text to display while searching for more results in a limited suggestions list
- **selectedItems** `T[]`
  The items that the base picker should currently display as selected. If this is provided then the picker will act as a controlled component.
- **styles** `IStyleFunctionOrObject<IBasePickerStyleProps, IBasePickerStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by styled() function.
- **"aria-label"** `string`
  Screen reader label to apply to an input element.
- **defaultVisibleValue** `string`
  The default value to be visible when the autofill first created. This is different than placeholder text because the placeholder text will disappear and re-appear. This text persists until deleted or changed.
- **componentRef** `IRefObject<IPickerItem>`
  Optional callback to access the IPickerItem interface. Use this instead of ref for accessing the public methods and properties of the component.

- **index** `number`
  Index number of the item in the array of picked items.
- **item** `T`
  The item of Type T (Persona, Tag, or any other custom item provided).
- **key** `string | number`
  Unique key for each picked item.
- **onItemChange** `(item: T, index: number) => void`
  Internal Use only, gives a callback to the renderer to call when an item has changed. This allows the base picker to keep track of changes in the items.
- **onRemoveItem** `() => void`
  Callback issued when the item is removed from the array of picked items.
- **removeButtonAriaLabel** `string`
  Aria-label for the picked item remove button.
- **selected** `boolean`
  Whether the picked item is selected or not.
- **className** `string`
  Optional className for the root element of the suggestion item.
- **componentRef** `IRefObject<ISuggestionsItem>`
  Optional callback to access the ISuggestionItem interface. Use this instead of ref for accessing the public methods and properties of the component.
- **id** `string`
  Unique id of the suggested item.
- **isSelectedOverride** `boolean`
  An override for the 'selected' property of the SuggestionModel.
- **onClick** `(ev: React.MouseEvent<HTMLButtonElement>) => void`
  Callback for when the user clicks on the suggestion.
- **onRemoveItem** `(ev: React.MouseEvent<HTMLButtonElement>) => void`
  Callback for when the item is removed from the array of suggested items.
- **removeButtonAriaLabel** `string`
  The ARIA label for the button to remove the suggestion from the list.
- **RenderSuggestion** `(item: T, suggestionItemProps: ISuggestionItemProps<T>) => JSX.Element`
  Optional renderer to override the default one for each type of picker.
- **showRemoveButton** `boolean`
  Whether the remove button should be rendered or not.
- **styles** `IStyleFunctionOrObject<ISuggestionsItemStyleProps, ISuggestionsItemStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **suggestionModel** `ISuggestionModel<T>`
  Individual suggestion object containing its properties.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **className** `string`
  The CSS className of the suggestions root.

- **componentRef** `IRefObject<ISuggestions<T>>`
  Optional callback to access the ISuggestions interface. Use this instead of ref for accessing the public methods and properties of the component.
- **createGenericItem** `() => void`
  The callback that should be called when the user attempts to use the input text as as item
- **forceResolveText** `string`
  The text that appears indicating to the use to force resolve the input
- **isLoading** `boolean`
  Used to indicate whether or not the suggestions are loading.
- **isMostRecentlyUsedVisible** `boolean`
  Indicates if a short list of recent suggestions should be shown.
- **isResultsFooterVisible** `boolean`
  Indicates if the text in resultsFooter or resultsFooterFull should be shown at the end of the suggestion list.
- **isSearching** `boolean`
  Used to indicate whether or not the component is searching for more results.
- **loadingText** `string`
  The text to display while the results are loading.
- **moreSuggestionsAvailable** `boolean`
  Used to indicate whether or not the user can request more suggestions. Dictates whether or not the searchForMore button is displayed.
- **mostRecentlyUsedHeaderText** `string`
  The text that should appear at the top of the most recently used box.
- **noResultsFoundText** `string`
  The text that should appear if no results are found when searching.
- **onGetMoreResults** `() => void`
  The callback that should be called when the user attempts to get more results
- **onRenderNoResultFound** `IRenderFunction<void>`
  How the "no result found" should look in the suggestion list.
- **onRenderSuggestion** `(props: T, suggestionItemProps: ISuggestionItemProps<T>) => JSX.Element`
  How the suggestion should look in the suggestion list.
- **onSuggestionClick** `(ev?: React.MouseEvent<HTMLElement>, item?: any, index?: number) => void`
  What should occur when a suggestion is clicked
- **onSuggestionRemove** `(ev?: React.MouseEvent<HTMLElement>, item?: T | IPersonaProps, index?: number)`
  Function to fire when one of the optional remove buttons on a suggestion is clicked.

TODO (adjective-object) remove IPersonaprops before the next major version bump

- **refocusSuggestions** `(keyCode: KeyCodes) => void`
  A function that resets focus to the expected item in the suggestion list
- **removeSuggestionAriaLabel** `string`
  An ARIA label to use for the buttons to remove individual suggestions.
- **resultsFooter** `(props: ISuggestionsProps<T>) => JSX.Element`
  A renderer that adds an element at the end of the suggestions list it has fewer items than resultsMaximumNumber.

- **resultsFooterFull** `(props: ISuggestionsProps<T>) => JSX.Element`
  A renderer that adds an element at the end of the suggestions list it has more items than resultsMaximumNumber.
- **resultsMaximumNumber** `number`
  Maximum number of suggestions to show in the full suggestion list.
- **searchErrorText** `string`
  The text that should appear if there is a search error.
- **searchForMoreText** `string`
  The text that appears indicating to the user that they can search for more results.
- **searchingText** `string`
  The text to display while searching for more results in a limited suggestions list.
- **showForceResolve** `() => boolean`
  The callback that should be called to see if the force resolve command should be shown
- **showRemoveButtons** `boolean`
  Indicates whether to show a button with each suggestion to remove that suggestion.
- **styles** `IStyleFunctionOrObject<any, any>`
  Call to provide customized styling that will layer on top of the variant rules.
- **suggestions** `ISuggestionModel<T>[]`
  The list of Suggestions that will be displayed
- **suggestionsAvailableAlertText** `string`
  Screen reader message to read when there are suggestions available.
- **suggestionsClassName** `string`
  The CSS className of the suggestions list
- **suggestionsContainerAriaLabel** `string`
  An ARIA label for the container that is the parent of the suggestions.
- **suggestionsHeaderText** `string`
  The text that appears at the top of the suggestions list.
- **suggestionsItemClassName** `string`
  The className of the suggestion item.
- **suggestionsListId** `string`
  The string that will be used as the suggestionsListId. Will be used by the BasePicker to keep track of the list for aria.
- **theme** `ITheme`
  Theme provided by High-Order Component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use a picker to quickly search for a few tags or files.
- Use a picker to manage a group of tags or files.

Examples

```
library(shiny)
library(shiny.fluent)

makeScript <- function(js) {
  htmltools::htmlDependency(
    name = "TagPickerExample",
    version = "0", # Not used.
    src = c(href = ""), # Not used.
    head = paste0("<script>", js, "</script>")
  )
}

ui <- function(id) {
  ns <- NS(id)
  tagList(
    makeScript("
      testTags = [
        'black',
        'blue',
        'brown',
        'cyan',
        'green',
        'magenta',
        'mauve',
        'orange',
        'pink',
        'purple',
        'red',
        'rose',
        'violet',
        'white',
        'yellow',
      ].map(item => ({ key: item, name: item }));

      function listContainsTagList(tag, tagList) {
        if (!tagList || !tagList.length || tagList.length === 0) {
          return false;
        }
        return tagList.some(compareTag => compareTag.key === tag.key);
      };

      function filterSuggestedTags(filterText, tagList) {
        return filterText
          ? testTags.filter(
              tag => tag.name.toLowerCase().indexOf(filterText.toLowerCase()) === 0 &&
               !listContainsTagList(tag, tagList),
            )
          : [];
      };
    "),
    textOutput(ns("selectedTags")),
```

```
    TagPicker(
      onResolveSuggestions = JS("filterSuggestedTags"),
      onEmptyInputFocus = JS(
     "function(tagList) { return testTags.filter(tag => !listContainsTagList(tag, tagList)); }"
      ),
      getTextFromItem = JS("function(item) { return item.text }"),
      pickerSuggestionsProps = list(
        suggestionsHeaderText = 'Suggested tags',
        noResultsFoundText = 'No color tags found'
      ),
      itemLimit = 2,
      onChange = JS(paste0(
        "function(selection) {",
        "  Shiny.setInputValue('", ns("selectedTags") ,"', JSON.stringify(selection));",
        "}"
      ))
    )
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$selectedTags <- renderText({
      if (is.null(input$selectedTags)) {
        "Select up to 2 colors below:"
      } else {
        paste(
          "You have selected:",
          paste(jsonlite::fromJSON(input$selectedTags)$name, collapse = ", ")
        )
      }
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Breadcrumb` *Breadcrumb*

Description

Breadcrumbs should be used as a navigational aid in your app or site. They indicate the current page's location within a hierarchy and help the user understand where they are in relation to the rest of that hierarchy. They also afford one-click access to higher levels of that hierarchy.

Breadcrumbs are typically placed, in horizontal form, under the masthead or navigation of an experience, above the primary content area.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Breadcrumb(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaLabel** `string`
  Aria label for the root element of the breadcrumb (which is a navigation landmark).
- **className** `string`
  Optional class for the root breadcrumb element.
- **componentRef** `IRefObject<IBreadcrumb>`
  Optional callback to access the IBreadcrumb interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dividerAs** `IComponentAs<IDividerAsProps>`
  Render a custom divider in place of the default chevron >
- **focusZoneProps** `IFocusZoneProps`
  Extra props for the root FocusZone.
- **items** `IBreadcrumbItem[]`
  Collection of breadcrumbs to render
- **maxDisplayedItems** `number`
  The maximum number of breadcrumbs to display before coalescing. If not specified, all breadcrumbs will be rendered.
- **onGrowData** `(data: IBreadcrumbData) => IBreadcrumbData | undefined`
  Method that determines how to group the length of the breadcrumb. Return undefined to never increase breadcrumb length.
- **onReduceData** `(data: IBreadcrumbData) => IBreadcrumbData | undefined`
  Method that determines how to reduce the length of the breadcrumb. Return undefined to never reduce breadcrumb length.
- **onRenderItem** `IRenderFunction<IBreadcrumbItem>`
  Custom render function for each breadcrumb item.
- **onRenderOverflowIcon** `IRenderFunction<IButtonProps>`
  Render a custom overflow icon in place of the default icon ...
- **overflowAriaLabel** `string`
  Aria label for the overflow button.
- **overflowIndex** `number`
  Optional index where overflow items will be collapsed. Defaults to 0.

- **styles** `IStyleFunctionOrObject<IBreadcrumbStyleProps, IBreadcrumbStyles>`
- **theme** `ITheme`
- **tooltipHostProps** `ITooltipHostProps`
  Extra props for the TooltipHost which wraps each breadcrumb item.
- **item** `IBreadcrumbItem`
  Breadcrumb item to left of the divider to be passed for custom rendering. For overflowed items, it will be last item in the list.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

items <- list(
  list(text = "Files", key = "Files", href = "#/page"),
  list(text = "Folder 1", key = "f1", href = "#/page"),
  list(text = "Folder 2", key = "f2", href = "#/page"),
  list(text = "Folder 3", key = "f3", href = "#/page"),
  list(text = "Folder 4 (non-clickable)", key = "f4"),
  list(text = "Folder 5", key = "f5", href = "#/page", isCurrentItem = TRUE)
)

ui <- function(id) {
  Breadcrumb(
    items = items,
    maxDisplayedItems = 3,
    ariaLabel = "Breadcrumb with items rendered as links",
    overflowAriaLabel = "More links"
  )
}
server <- function(id) {
  moduleServer(id, function(input, output, session) { })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Calendar` *Calendar*

Description

The calendar control lets people select and view a single date or a range of dates in their calendar. It's made up of 3 separate views: the month view, year view, and decade view.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Calendar(...)

Calendar.shinyInput(inputId, ..., value = shiny.react::JS("new Date()"))

updateCalendar.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **allFocusable** `boolean`
  Allows all dates and buttons to be focused, including disabled ones
- **autoNavigateOnSelection** `boolean`
  Whether the month view should automatically navigate to the next or previous date range depending on the selected date. If this property is set to true and the currently displayed month is March 2017, if the user clicks on a day outside the month, i.e., April 1st, the picker will automatically navigate to the month of April.
- **className** `string`
  Optional class name to add to the root element.
- **componentRef** `IRefObject<ICalendar>`
  Optional callback to access the ICalendar interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dateRangeType** `DateRangeType`
  The date range type indicating how many days should be selected as the user selects days
- **dateTimeFormatter** `ICalendarFormatDateCallbacks`
  Apply additional formating to dates, for example localized date formatting.

- **firstDayOfWeek** `DayOfWeek`
  The first day of the week for your locale.
- **firstWeekOfYear** `FirstWeekOfYear`
  Defines when the first week of the year should start, FirstWeekOfYear.FirstDay, FirstWeekOfYear.FirstFullWeek or FirstWeekOfYear.FirstFourDayWeek are the possible values
- **highlightCurrentMonth** `boolean`
  Whether the month picker should highlight the current month
- **highlightSelectedMonth** `boolean`
  Whether the month picker should highlight the selected month
- **isDayPickerVisible** `boolean`
  Whether the day picker is shown beside the month picker or hidden.
- **isMonthPickerVisible** `boolean`
  Whether the month picker is shown beside the day picker or hidden.
- **maxDate** `Date`
  If set the Calendar will not allow navigation to or selection of a date later than this value.
- **minDate** `Date`
  If set the Calendar will not allow navigation to or selection of a date earlier than this value.
- **navigationIcons** `ICalendarIconStrings`
  Customize navigation icons using ICalendarIconStrings
- **onDismiss** `() => void`
  Callback issued when calendar is closed
- **onSelectDate** `(date: Date, selectedDateRangeArray?: Date[]) => void`
  Callback issued when a date is selected
- **restrictedDates** `Date[]`
  If set the Calendar will not allow selection of dates in this array.
- **selectDateOnClick** `boolean`
  When clicking on "Today", select the date and close the calendar.
- **shouldFocusOnMount** `boolean`
  This property has been removed at 0.80.0 in place of the `focus` method, to be removed \@ 1.0.0.
- **showCloseButton** `boolean`
  Whether the close button should be shown or not
- **showGoToToday** `boolean`
  Whether the "Go to today" link should be shown or not
- **showMonthPickerAsOverlay** `boolean`
  Show month picker on top of date picker when visible.
- **showSixWeeksByDefault** `boolean`
  Whether the calendar should show 6 weeks by default.
- **showWeekNumbers** `boolean`
  Whether the calendar should show the week number (weeks 1 to 53) before each week row
- **strings** `ICalendarStrings | null`
  Localized strings to use in the Calendar

- **today** `Date`
  Value of today. If null, current time in client machine will be used.
- **value** `Date`
  Default value of the Calendar, if any
- **workWeekDays** `DayOfWeek[]`
  The days that are selectable when dateRangeType is WorkWeek. If dateRangeType is not WorkWeek this property does nothing.
- **yearPickerHidden** `boolean`
  Whether the year picker is enabled

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Don't break the control apart.
- Include an up and down arrow for navigating between time ranges and a chevron to make the calendar collapsible.

Content:

- Use the following format for dates: month, day, year, as in July 31, 2016. When space is limited, use numbers and slashes for dates if the code supports that format and automatically displays the appropriate date format for different locales. For example, 2/16/19.
- Don't use ordinal numbers (such as 1st, 12th, or 23rd) to indicate a date.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    Calendar.shinyInput(ns("calendar"), value = "2020-06-25T22:00:00.000Z"),
    textOutput(ns("calendarValue")),
    h3("If `value` is missing, default to system date"),
    Calendar.shinyInput(ns("calendar2")),
    textOutput(ns("calendarDefault")),
    h3("If `value` is NULL, also default to system date"),
    Calendar.shinyInput(ns("calendar3"), value = NULL),
    textOutput(ns("calendarNull"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
```

```
    output$calendarValue <- renderText({
      sprintf("Value: %s", input$calendar)
    })
    output$calendarDefault <- renderText({
      sprintf("Value: %s", input$calendar2)
    })
    output$calendarNull <- renderText({
      sprintf("Value: %s", input$calendar3)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Callout` *Callout*

Description

A callout is an anchored tip that can be used to teach people or guide them through the app without blocking them.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Callout(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **alignTargetEdge** `boolean`
  If true the positioning logic will prefer to flip edges rather than to nudge the rectangle to fit within bounds, thus making sure the element aligns perfectly with target's alignment edge
- **ariaDescribedBy** `string`
  Defines the element id referencing the element containing the description for the callout.
- **ariaLabel** `string`
  Accessible label text for callout.
- **ariaLabelledBy** `string`
  Defines the element id referencing the element containing label text for callout.

- **backgroundColor** `string`
  The background color of the Callout in hex format ie. #ffffff.
- **beakWidth** `number`
  The width of the beak.
- **bounds** `IRectangle | ((target?: Target, targetWindow?: Window) => IRectangle | undefined)`
  The bounding rectangle (or callback that returns a rectangle) for which the contextual menu can appear in.
- **calloutMaxHeight** `number`
  Set max height of callout When not set the callout will expand with contents up to the bottom of the screen
- **calloutMaxWidth** `number`
  Custom width for callout including borders. If value is 0, no width is applied.
- **calloutWidth** `number`
  Custom width for callout including borders. If value is 0, no width is applied.
- **className** `string`
  CSS class to apply to the callout.
- **coverTarget** `boolean`
  If true the position returned will have the menu element cover the target. If false then it will position next to the target;
- **directionalHint** `DirectionalHint`
  How the element should be positioned
- **directionalHintFixed** `boolean`
  If true the position will not change sides in an attempt to fit the callout within bounds. It will still attempt to align it to whatever bounds are given.
- **directionalHintForRTL** `DirectionalHint`
  How the element should be positioned in RTL layouts. If not specified, a mirror of the `directionalHint` alignment edge will be used instead. This means that `DirectionalHint.BottomLeft` will change to `DirectionalHint.BottomRight` but `DirectionalHint.LeftAuto` will not change.
- **doNotLayer** `boolean`
  If true do not render on a new layer. If false render on a new layer.
- **finalHeight** `number`
  Specify the final height of the content. To be used when expanding the content dynamically so that callout can adjust its position.
- **gapSpace** `number`
  The gap between the Callout and the target
- **hidden** `boolean`
  If specified, renders the Callout in a hidden state. Use this flag, rather than rendering a callout conditionally based on visibility, to improve rendering performance when it becomes visible. Note: When callout is hidden its content will not be rendered. It will only render once the callout is visible.
- **hideOverflow** `boolean`
  Manually set OverflowYHidden style prop to true on calloutMain element A variety of callout load animations will need this to hide the scollbar that can appear

- **isBeakVisible** `boolean`
  If true then the beak is visible. If false it will not be shown.
- **layerProps** `ILayerProps`
  Optional props to pass to the Layer component hosting the panel.
- **minPagePadding** `number`
  The minimum distance the callout will be away from the edge of the screen.
- **onDismiss** `(ev?: any) => void`
  Callback when the Callout tries to close.
- **onLayerMounted** `() => void`
  Optional callback when the layer content has mounted.
- **onPositioned** `(positions?: ICalloutPositionedInfo) => void`
  Optional callback that is called once the callout has been correctly positioned.
- **onRestoreFocus** `(options: { originalElement?: HTMLElement | Window; containsFocus: boolean; }) => v`
  Called when the component is unmounting, and focus needs to be restored. Argument passed down contains two variables, the element that the underlying popup believes focus should go to * and whether or not the popup currently contains focus. If this is provided, focus will not be restored automatically, you'll need to call originalElement.focus()
- **onScroll** `() => void`
  Callback when the Callout body is scrolled.
- **preventDismissOnLostFocus** `boolean`
  If true then the callout will not dismiss when it loses focus
- **preventDismissOnResize** `boolean`
  If true then the callout will not dismiss on resize
- **preventDismissOnScroll** `boolean`
  If true then the callout will not dismiss on scroll
- **role** `string`
  Aria role assigned to the callout (Eg. dialog, alertdialog).
- **setInitialFocus** `boolean`
  If true then the callout will attempt to focus the first focusable element that it contains. If it doesn't find an element, no focus will be set and the method will return false. This means that it's the contents responsibility to either set focus or have focusable items.
- **shouldRestoreFocus** `boolean`
  If true, when this component is unmounted, focus will be restored to the element that had focus when the component first mounted.
- **shouldUpdateWhenHidden** `boolean`
  If true, the component will be updated even when hidden=true. Note that this would consume resources to update even though nothing is being shown to the user. This might be helpful though if your updates are small and you want the callout to be revealed fast to the user when hidden is set to false.
- **style** `React.CSSProperties`
  CSS style to apply to the callout.

If you set `overflowY` in this object, it provides a performance optimization by preventing Popup (underlying component of Callout) from calculating whether it needs a scroll bar.

- **styles** `IStyleFunctionOrObject<ICalloutContentStyleProps, ICalloutContentStyles>`
  Optional styles for the component.
- **target** `Target`
  The target that the Callout should try to position itself based on. It can be either an Element a querySelector string of a valid Element or a MouseEvent. If MouseEvent is given then the origin point of the event will be used.
- **theme** `ITheme`
  Optional theme for component

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Don't use a callout to ask for action confirmation; use a dialog instead.
- Place a callout near the object being described. At the pointer's tail or head, if possible.
- Don't use large, unformatted blocks of text in your callout. They're difficult to read and overwhelming.
- Don't block important UI with the placement of your callout. It's a poor user experience that will lead to frustration.
- Don't open a callout from within another callout.
- Don't show callouts on hidden elements.
- Don't overuse callouts. Too many callouts opening automatically can be perceived as interrupting someone's workflow.
- For a particularly complex concept that needs explanation, place an info icon (`iconClassNames.info`) next to the concept to indicate there's more helpful information available. When someone hovers over or selects the icon, the callout should appear.

Content:

- Because the content inside of a callout isn't always visible, don't put required information in a callout.
- Short sentences or sentence fragments are best.
- Don't use obvious tip text or text that simply repeats what is already on the screen. Limit the information inside of a callout to supplemental information.
- When additional context or a more advanced description is necessary, consider placing a link to "Learn more" at the bottom of the callout. When clicked, open the additional content in a new window or panel.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
```

```
  div(
    DefaultButton.shinyInput(ns("toggleCallout"), text = "Toggle Callout"),
    reactOutput(ns("callout"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    show <- reactiveVal(FALSE)
    observeEvent(input$toggleCallout, show(!show()))
    output$callout <- renderReact({
      if (show()) {
        Callout(
          tags$div(
            style = "margin: 10px",
            "Callout contents"
          )
        )
      }
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

Checkbox *Checkbox*

Description

Check boxes (`Checkbox`) give people a way to select one or more items from a group, or switch between two mutually exclusive options (checked or unchecked, on or off).

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Checkbox(...)

Checkbox.shinyInput(inputId, ..., value = defaultValue)

updateCheckbox.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **ariaDescribedBy** `string`
  ID for element that provides extended information for the checkbox.
- **ariaLabel** `string`
  Accessible label for the checkbox.
- **ariaLabelledBy** `string`
  ID for element that contains label information for the checkbox.
- **ariaPositionInSet** `number`
  The position in the parent set (if in a set) for aria-posinset.
- **ariaSetSize** `number`
  The total size of the parent set (if in a set) for aria-setsize.
- **boxSide** `'start' | 'end'`
  Allows you to set the checkbox to be at the before (start) or after (end) the label.
- **checked** `boolean`
  Checked state. Mutually exclusive to "defaultChecked". Use this if you control the checked state at a higher level and plan to pass in the correct value based on handling onChange events and re-rendering.
- **checkmarkIconProps** `IIconProps`
  Custom icon props for the check mark rendered by the checkbox
- **className** `string`
  Additional class name to provide on the root element, in addition to the ms-Checkbox class.
- **componentRef** `IRefObject<ICheckbox>`
  Optional callback to access the ICheckbox interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultChecked** `boolean`
  Default checked state. Mutually exclusive to "checked". Use this if you want an uncontrolled component, and want the Checkbox instance to maintain its own state.
- **defaultIndeterminate** `boolean`
  Optional uncontrolled indeterminate visual state for checkbox. Setting indeterminate state takes visual precedence over checked or defaultChecked props given but does not affect checked state. This is not a toggleable state. On load the checkbox will receive indeterminate visual state and after the user's first click it will be removed exposing the true state of the checkbox.
- **disabled** `boolean`
  Disabled state of the checkbox.

- **indeterminate** `boolean`
  Optional controlled indeterminate visual state for checkbox. Setting indeterminate state takes visual precedence over checked or defaultChecked props given but does not affect checked state. This should not be a toggleable state. On load the checkbox will receive indeterminate visual state and after the first user click it should be removed by your supplied onChange callback function exposing the true state of the checkbox.
- **inputProps** `React.ButtonHTMLAttributes<HTMLElement | HTMLButtonElement>`
  Optional input props that will be mixed into the input element, *before* other props are applied. This allows you to extend the input element with additional attributes, such as data-automation-id needed for automation. Note that if you provide, for example, "disabled" as well as "inputProps.disabled", the former will take precedence over the later.
- **keytipProps** `IKeytipProps`
  Optional keytip for this checkbox
- **label** `string`
  Label to display next to the checkbox.
- **onChange** `(ev?: React.FormEvent<HTMLElement | HTMLInputElement>, checked?: boolean) => void`
  Callback that is called when the checked value has changed.
- **onRenderLabel** `IRenderFunction<ICheckboxProps>`
  Custom render function for the label.
- **styles** `IStyleFunctionOrObject<ICheckboxStyleProps, ICheckboxStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a single check box when there's only one selection to make or choice to confirm. Selecting a blank check box selects it. Selecting it again clears the check box.
- Use multiple check boxes when one or more options can be selected from a group. Unlike radio buttons, selecting one check box will not clear another check box.

Content:

- Separate two groups of check boxes with headings rather than positioning them one after the other.
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.
- Don't use end punctuation (unless the check box label absolutely requires multiple sentences).
- Use a sentence fragment for the label, rather than a full sentence.
- Make it easy for people to understand what will happen if they select or clear a check box.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    Checkbox.shinyInput(ns("checkbox"), value = FALSE),
    textOutput(ns("checkboxValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$checkboxValue <- renderText({
      sprintf("Value: %s", input$checkbox)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

ChoiceGroup *ChoiceGroup*

Description

Radio buttons (ChoiceGroup) let people select a single option from two or more choices.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ChoiceGroup(...)

ChoiceGroup.shinyInput(inputId, ..., value = defaultValue)

updateChoiceGroup.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **componentRef** IRefObject<IChoiceGroupOption>
  Optional callback to access the IChoiceGroup interface. Use this instead of ref for accessing the public methods and properties of the component.
- **focused** boolean
  Indicates if the ChoiceGroupOption should appear focused, visually
- **name** string
  This value is used to group each ChoiceGroupOption into the same logical ChoiceGroup
- **onBlur** (ev: React.FocusEvent<HTMLElement>, props?: IChoiceGroupOption) => void
  A callback for receiving a notification when the choice has lost focus.
- **onChange** (evt?: React.FormEvent<HTMLElement | HTMLInputElement>, props?: IChoiceGroupOption) =>
  A callback for receiving a notification when the choice has been changed.
- **onFocus** (ev?: React.FocusEvent<HTMLElement | HTMLInputElement>, props?: IChoiceGroupOption) => vo
  A callback for receiving a notification when the choice has received focus.
- **required** boolean
  If true, it specifies that an option must be selected in the ChoiceGroup before submitting the form
- **theme** ITheme
  Theme (provided through customization.)
- **ariaLabelledBy** string
  ID of an element to use as the aria label for this ChoiceGroup.
- **componentRef** IRefObject<IChoiceGroup>
  Optional callback to access the IChoiceGroup interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultSelectedKey** string | number
  The key of the option that will be initially checked.
- **label** string
  Descriptive label for the choice group.
- **onChange** (ev?: React.FormEvent<HTMLElement | HTMLInputElement>, option?: IChoiceGroupOption) =>
  A callback for receiving a notification when the choice has been changed.
- **onChanged** (option: IChoiceGroupOption, evt?: React.FormEvent<HTMLElement | HTMLInputElement>) =>
  Deprecated and will be removed by 07/17/2017. Use onChange instead.
- **options** IChoiceGroupOption[]
  The options for the choice group.

- **selectedKey** `string | number`
  The key of the selected option. If you provide this, you must maintain selection state by observing onChange events and passing a new value in when changed.
- **styles** `IStyleFunctionOrObject<IChoiceGroupStyleProps, IChoiceGroupStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization).

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use radio buttons when there are two to seven options, you have enough screen space, and the options are important enough to be a good use of that screen space.
- If there are more than seven options, use a drop-down menu instead.
- To give people a way to select more than one option, use check boxes instead.
- If a default option is recommended for most people in most situations, use a drop-down menu instead.
- Align radio buttons vertically instead of horizontally, if possible. Horizontal alignment is harder to read and localize. If there are only two mutually exclusive options, combine them into a single check box or toggle. For example, use a check box for "I agree" statements instead of radio buttons for "I agree" and "I disagree".

Content:

- List the options in a logical order, such as most likely to be selected to least, simplest operation to most complex, or least risk to most. Listing options in alphabetical order isn't recommended because the order will change when the text is localized.
- Select the safest (to prevent loss of data or system access), most secure, and most private option as the default. If safety and security aren't factors, select the most likely or convenient option.
- Use a phrase for the label, rather than a full sentence.
- Make sure to give people the option to not make a choice. For example, include a "None" option.

Examples

```
library(shiny)
library(shiny.fluent)

options <- list(
  list(key = "A", text = "Option A"),
  list(key = "B", text = "Option B"),
  list(key = "C", text = "Option C")
)
```

```
ui <- function(id) {
  ns <- NS(id)
  div(
    ChoiceGroup.shinyInput(ns("choice"), value = "B", options = options),
    textOutput(ns("groupValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$groupValue <- renderText({
      sprintf("Value: %s", input$choice)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Coachmark` *Coachmark*

Description

Coach marks (`Coachmark`) are used to draw a person's attention to parts of the UI and increase engagement with those elements. A teaching bubble appears on hover or selection of the coach mark.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Coachmark(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaAlertText** `string`
  Text to announce to screen reader / narrator when Coachmark is displayed
- **ariaDescribedBy** `string`
  Defines the element id referencing the element containing the description for the Coachmark.
- **ariaDescribedByText** `string`
  Defines the text content for the ariaDescribedBy element

- **ariaLabelledBy** `string`
  Defines the element id referencing the element containing label text for Coachmark.
- **ariaLabelledByText** `string`
  Defines the text content for the ariaLabelledBy element
- **beaconColorOne** `string`
  Beacon color one.
- **beaconColorTwo** `string`
  Beacon color two.
- **beakHeight** `number`
  The height of the Beak component.
- **beakWidth** `number`
  The width of the Beak component.
- **className** `string`
  If provided, additional class name to provide on the root element.
- **collapsed** `boolean`
  The starting collapsed state for the Coachmark. Use `isCollapsed` instead.
- **color** `string`
  Color of the Coachmark/TeachingBubble.
- **componentRef** `IRefObject<ICoachmark>`
  Optional callback to access the ICoachmark interface. Use this instead of ref for accessing the public methods and properties of the component.
- **delayBeforeCoachmarkAnimation** `number`
  Delay in milliseconds before Coachmark animation appears.
- **delayBeforeMouseOpen** `number`
  Delay before allowing mouse movements to open the Coachmark.
- **height** `number`
  The height of the Coachmark.
- **isCollapsed** `boolean`
  The starting collapsed state for the Coachmark.
- **isPositionForced** `boolean`
  Whether or not to force the Coachmark/TeachingBubble content to fit within the window bounds.
- **mouseProximityOffset** `number`
  The distance in pixels the mouse is located before opening up the Coachmark.
- **onAnimationOpenEnd** `() => void`
  Callback when the opening animation completes.
- **onAnimationOpenStart** `() => void`
  Callback when the opening animation begins.
- **onDismiss** `(ev?: any) => void`
  Callback when the Coachmark tries to close.
- **onMouseMove** `(e: MouseEvent) => void`
  Callback to run when the mouse moves.

- **persistentBeak** `boolean`
  If true then the Coachmark beak (arrow pointing towards target) will always be visible as long as Coachmark is visible
- **positioningContainerProps** `IPositioningContainerProps`
  Props to pass to the PositioningContainer component. Specify the `directionalHint` to indicate on which edge the Coachmark/TeachingBubble should be positioned.
- **preventDismissOnLostFocus** `boolean`
  If true then the Coachmark will not dismiss when it loses focus
- **preventFocusOnMount** `boolean`
  If true then focus will not be set to the Coachmark when it mounts. Useful in cases where focus on coachmark is causing other components in page to dismiss upon losing focus.
- **styles** `IStyleFunctionOrObject<ICoachmarkStyleProps, ICoachmarkStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **target** `HTMLElement | string | null`
  The target that the Coachmark should try to position itself based on.
- **teachingBubbleRef** `ITeachingBubble`
  Ref for TeachingBubble
- **theme** `ITheme`
  Theme provided by higher order component.
- **width** `number`
  The width of the Coachmark.
- **ariaDescribedBy** `string`
  Defines the element id referencing the element containing the description for the positioningContainer.
- **ariaLabel** `string`
  Accessible label text for positioningContainer.
- **ariaLabelledBy** `string`
  Defines the element id referencing the element containing label text for positioningContainer.
- **backgroundColor** `string`
  The background color of the positioningContainer in hex format ie. #ffffff.
- **bounds** `IRectangle`
  The bounding rectangle for which the contextual menu can appear in.
- **className** `string`
  CSS class to apply to the positioningContainer.
- **componentRef** `IRefObject<IPositioningContainer>`
  All props for your component are to be defined here.
- **coverTarget** `boolean`
  If true the position returned will have the menu element cover the target. If false then it will position next to the target;
- **directionalHint** `DirectionalHint`
  How the element should be positioned
- **directionalHintFixed** `boolean`
  If true the position will not change sides in an attempt to fit the positioningContainer within bounds. It will still attempt to align it to whatever bounds are given.

- **directionalHintForRTL** `DirectionalHint`
  How the element should be positioned in RTL layouts. If not specified, a mirror of `directionalHint` will be used instead
- **doNotLayer** `boolean`
  If true do not render on a new layer. If false render on a new layer.
- **finalHeight** `number`
  Specify the final height of the content. To be used when expanding the content dynamically so that positioningContainer can adjust its position.
- **minPagePadding** `number`
  The minimum distance the positioningContainer will be away from the edge of the screen.
- **offsetFromTarget** `number`
  The gap between the positioningContainer and the target
- **onDismiss** `(ev?: any) => void`
  Callback when the positioningContainer tries to close.
- **onLayerMounted** `() => void`
  Optional callback when the layer content has mounted.
- **onPositioned** `(positions?: IPositionedData) => void`
  Optional callback that is called once the positioningContainer has been correctly positioned.
- **positioningContainerMaxHeight** `number`
  Set max height of positioningContainer When not set the positioningContainer will expand with contents up to the bottom of the screen
- **positioningContainerWidth** `number`
  Custom width for positioningContainer including borders. If value is 0, no width is applied.
- **preventDismissOnScroll** `boolean`
  If true then the onClose will not not dismiss on scroll
- **role** `string`
  Aria role assigned to the positioningContainer (Eg. dialog, alertdialog).
- **setInitialFocus** `boolean`
  If true then the positioningContainer will attempt to focus the first focusable element that it contains. If it doesn't find an element, no focus will be set and the method will return false. This means that it's the contents responsibility to either set focus or have focusable items.
- **target** `HTMLElement | string | MouseEvent | Point | null`
  The target that the positioningContainer should try to position itself based on. It can be either an HTMLElement a querySelector string of a valid HTMLElement or a MouseEvent. If MouseEvent is given then the origin point of the event will be used.
- **targetPoint** `Point`
  Point used to position the positioningContainer. Deprecated, use `target` instead.
- **useTargetPoint** `boolean`
  If true use a point rather than rectangle to position the positioningContainer. For example it can be used to position based on a click.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Only one coach mark and teaching bubble combo should be displayed at a time.
- Coach marks can be standalone or sequential. Sequential coach marks should be used sparingly to walk through complex multistep interactions. It's recommended that a sequence of coach marks doesn't exceed three steps.
- Coach marks are designed to only hold teaching bubbles.
- Coach mark size, color, and animation shouldn't be altered.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  tagList(
    uiOutput(ns("coachmark")),
    DefaultButton.shinyInput(ns("toggleCoachmark"),
      id = "target", text = "Toggle coachmark"
    )
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    coachmarkVisible <- reactiveVal(FALSE)
    observeEvent(input$toggleCoachmark, coachmarkVisible(!coachmarkVisible()))
    observeEvent(input$hideCoachmark, coachmarkVisible(FALSE))
    output$coachmark <- renderUI({
      if (coachmarkVisible()) Coachmark(
        target = "#target",
        TeachingBubbleContent(
          hasCloseButton = TRUE,
          onDismiss = triggerEvent(ns("hideCoachmark")),
          headline = "Example title",
          primaryButtonProps = list(text = "Try it"),
          secondaryButtonProps = list(text = "Try it again"),
          "Welcome to the land of coachmarks!"
        )
      )
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`ColorPicker` *ColorPicker*

Description

The color picker (`ColorPicker`) is used to browse through and select colors. By default, it lets people navigate through colors on a color spectrum; or specify a color in either Red-Green-Blue (RGB); or alpha color code; or Hexadecimal textboxes.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ColorPicker(...)

ColorPicker.shinyInput(inputId, ..., value = defaultValue)

updateColorPicker.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **alphaLabel** `string`
  Label for the alpha textfield.
- **alphaSliderHidden** `boolean`
  Whether to hide the alpha (or transparency) slider and text field.
- **alphaType** `'alpha' | 'transparency' | 'none'`
  `alpha` (the default) means display a slider and text field for editing alpha values. `transparency` also displays a slider and text field but for editing transparency values. `none` hides these controls.

Alpha represents the opacity of the color, whereas transparency represents the transparentness of the color: i.e. a 30% transparent color has 70% opaqueness.

- **blueLabel** `string`
  Label for the blue text field.
- **className** `string`
  Additional CSS class(es) to apply to the ColorPicker.
- **color** `IColor | string`
  Object or CSS-compatible string to describe the color.
- **componentRef** `IRefObject<IColorPicker>`
  Gets the component ref.
- **greenLabel** `string`
  Label for the green text field.
- **hexLabel** `string`
  Label for the hex text field.
- **onChange** `(ev: React.SyntheticEvent<HTMLElement>, color: IColor) => void`
  Callback for when the user changes the color. (Not called when the color is changed via props.)
- **redLabel** `string`
  Label for the red text field.
- **showPreview** `boolean`
  Whether to show color preview box.
- **strings** `IColorPickerStrings`
  Labels for elements within the ColorPicker. Defaults are provided in English only.
- **styles** `IStyleFunctionOrObject<IColorPickerStyleProps, IColorPickerStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization).
- **ariaDescription** `string`
  Detailed description for how to use the color rectangle. Moving the thumb horizontally adjusts saturation and moving it vertically adjusts value (essentially, brightness).
- **ariaLabel** `string`
  Label of the ColorRectangle for the benefit of screen reader users.
- **ariaValueFormat** `string`
  Format string for the color rectangle's current value as read by screen readers. The string must include descriptions and two placeholders for the current values: `{0}` for saturation and `{1}` for value/brightness.
- **className** `string`
  Additional CSS class(es) to apply to the ColorRectangle.
- **color** `IColor`
  Current color of the rectangle.
- **componentRef** `IRefObject<IColorRectangle>`
  Gets the component ref.
- **minSize** `number`
  Minimum width and height.

- **onChange** `(ev: React.MouseEvent | React.KeyboardEvent, color: IColor) => void`
  Callback for when the color changes.
- **styles** `IStyleFunctionOrObject<IColorRectangleStyleProps, IColorRectangleStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization).
- **ariaLabel** `string`
  Label of the ColorSlider for the benefit of screen reader users.
- **className** `string`
  Additional CSS class(es) to apply to the ColorSlider.
- **componentRef** `IRefObject<IColorSlider>`
  Gets the component ref.
- **isAlpha** `boolean`
  If true, the slider represents an alpha slider and will display a gray checkered pattern in the background. Otherwise, the slider represents a hue slider.
- **maxValue** `number`
  Maximum value of the slider.
- **minValue** `number`
  Minimum value of the slider.
- **onChange** `(event: React.MouseEvent | React.KeyboardEvent, newValue?: number) => void`
  Callback issued when the value changes.
- **overlayColor** `string`
  Hex color to use when rendering an alpha or transparency slider's overlay, *without* the #.
- **overlayStyle** `React.CSSProperties`
  Custom style for the overlay element.
- **styles** `IStyleFunctionOrObject<IColorSliderStyleProps, IColorSliderStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization).
- **thumbColor** `string`
  CSS-compatible string for the color of the thumb element.
- **type** `'hue' | 'alpha' | 'transparency'`
  Type of slider to display.
- **value** `number`
  Current value of the slider.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    ColorPicker.shinyInput(ns("color"), value = "#00FF01"),
    textOutput(ns("colorValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$colorValue <- renderText({
      sprintf("Value: %s", input$color)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

ComboBox *ComboBox*

Description

A combo box (ComboBox) combines a text field and a drop-down menu, giving people a way to select an option from a list or enter their own choice.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ComboBox(...)

VirtualizedComboBox(...)

ComboBox.shinyInput(inputId, ..., value = defaultValue)

updateComboBox.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **allowFreeform** `boolean`
  Whether the ComboBox is free form, meaning that the user input is not bound to provided options. Defaults to false.
- **ariaDescribedBy** `string`
  Optional prop to add a string id that can be referenced inside the aria-describedby attribute
- **autoComplete** `'on' | 'off'`
  Whether the ComboBox auto completes. As the user is inputing text, it will be suggested potential matches from the list of options. If the combo box is expanded, this will also scroll to the suggested option, and give it a selected style.
- **autofill** `IAutofillProps`
  The AutofillProps to be passed into the Autofill component inside combobox
- **buttonIconProps** `IIconProps`
  The IconProps to use for the button aspect of the combobox
- **caretDownButtonStyles** `Partial<IButtonStyles>`
  Styles for the caret down button.
- **comboBoxOptionStyles** `Partial<IComboBoxOptionStyles>`
  Default styles that should be applied to ComboBox options, in case an option does not come with user-defined custom styles
- **componentRef** `IRefObject<IComboBox>`
  Optional callback to access the IComboBox interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dropdownMaxWidth** `number`
  Custom max width for dropdown
- **dropdownWidth** `number`
  Custom width for dropdown (unless useComboBoxAsMenuWidth is undefined or false)
- **getClassNames** `(theme: ITheme, isOpen: boolean, disabled: boolean, required: boolean, focused: bool`
  Custom function for providing the classNames for the ComboBox. Can be used to provide all styles for the component instead of applying them on top of the default styles.
- **iconButtonProps** `IButtonProps`
  Optional iconButton props on combo box
- **isButtonAriaHidden** `boolean`
  Sets the 'aria-hidden' attribute on the ComboBox's button element instructing screen readers how to handle the element. This element is hidden by default because all functionality is handled by the input element and the arrow button is only meant to be decorative.

- **keytipProps** `IKeytipProps`
  Optional keytip for this combo box
- **multiSelectDelimiter** `string`
  When multiple items are selected, this will be used to separate values in the combobox input.
- **onChange** `(event: React.FormEvent<IComboBox>, option?: IComboBoxOption, index?: number, value?: s`
  Callback issued when either: 1) the selected option changes 2) a manually edited value is submitted. In this case there may not be a matched option if allowFreeform is also true (and hence only value would be true, the other parameter would be null in this case)
- **onItemClick** `(event: React.FormEvent<IComboBox>, option?: IComboBoxOption, index?: number) => void`
  Callback issued when a ComboBox item is clicked.
- **onMenuDismiss** `() => void`
  Function that gets invoked before the menu gets dismissed
- **onMenuDismissed** `() => void`
  Function that gets invoked when the ComboBox menu is dismissed
- **onMenuOpen** `() => void`
  Function that gets invoked when the ComboBox menu is launched
- **onPendingValueChanged** `(option?: IComboBoxOption, index?: number, value?: string) => void`
  Callback issued when the user changes the pending value in ComboBox. This will be called any time the component is updated and there is a current pending value. Option, index, and value will all be undefined if no change has taken place and the previously entered pending value is still valid.
- **onRenderLabel** `IRenderFunction<IOnRenderComboBoxLabelProps>`
  Custom render function for the label text.
- **onRenderLowerContent** `IRenderFunction<IComboBoxProps>`
  Add additional content below the callout list.
- **onRenderUpperContent** `IRenderFunction<IComboBoxProps>`
  Add additional content above the callout list.
- **onResolveOptions** `(options: IComboBoxOption[]) => IComboBoxOption[] | PromiseLike<IComboBoxOption[`
  Callback issued when the options should be resolved, if they have been updated or if they need to be passed in the first time
- **onScrollToItem** `(itemIndex: number) => void`
  Callback issued when the ComboBox requests the list to scroll to a specific element
- **options** `IComboBoxOption[]`
  Collection of options for this ComboBox
- **persistMenu** `boolean`
  Menu will not be created or destroyed when opened or closed, instead it will be hidden. This will improve perf of the menu opening but could potentially impact overall perf by having more elements in the dom. Should only be used when perf is important. Note: This may increase the amount of time it takes for the comboBox itself to mount.
- **scrollSelectedToTop** `boolean`
  When options are scrollable the selected option is positioned at the top of the callout when it is opened (unless it has reached the end of the scrollbar).

- **shouldRestoreFocus** `boolean`
  When specified, determines whether the callout (the menu which drops down) should restore the focus after being dismissed or not. If false, then the menu will not try to set focus to whichever element had focus before the menu was opened.
- **styles** `Partial<IComboBoxStyles>`
  Custom styles for this component
- **text** `string`
  Value to show in the input, does not have to map to a combobox option
- **theme** `ITheme`
  Theme provided by HOC.
- **useComboBoxAsMenuWidth** `boolean`
  Whether to use the ComboBoxes width as the menu's width
- **multiselectAccessibleText** `string`
  Accessible text for label when combobox is multiselected.
- **props** `IComboBoxProps`
  Props to render the combobox.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a combo box when there are multiple choices that can be collapsed under one title, when the list of items is long, or when space is constrained.

Content:

- Use single words or shortened statements as options.
- Don't use punctuation at the end of options.

Examples

```
library(shiny)
library(shiny.fluent)

options <- list(
  list(key = "A", text = "Option A"),
  list(key = "B", text = "Option B"),
  list(key = "C", text = "Option C")
)

ui <- function(id) {
  ns <- NS(id)
  div(
    ComboBox.shinyInput(ns("combo"), value = list(text = "some text"),
      options = options, allowFreeform = TRUE
```

```
    ),
    textOutput(ns("comboValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$comboValue <- renderText({
      sprintf("Value: %s", input$combo$text)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

CommandBar *CommandBar*

Description

CommandBar is a surface that houses commands that operate on the content of the window, panel, or parent region it resides above. CommandBars are one of the most visible and recognizable ways to surface commands, and can be an intuitive method for interacting with content on the page; however, if overloaded or poorly organized, they can be difficult to use and hide valuable commands from your user. CommandBars can also display a search box for finding content, hold simple commands as well as menus, or display the status of ongoing actions.

Commands should be sorted in order of importance, from left-to-right or right-to-left depending on the culture. Secondarily, organize commands in logical groupings for easier recall. CommandBars work best when they display no more than 5-7 commands. This helps users quickly find your most valuable features. If you need to show more commands, consider using the overflow menu. If you need to render status or viewing controls, these go on the right side of the CommandBar (or left side if in a left-to-right experience). Do not display more than 2-3 items on the right side as it will make the overall CommandBar difficult to parse.

All command items should have an icon and a label. Commands can render as labels only as well. In smaller widths, commands can just use icon only, but only for the most recognizable and frequently used commands. All other commands should go into an overflow where text labels can be shown.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
CommandBar(...)

CommandBar.shinyInput(inputId, ..., itemValueGetter = function(el) el$key)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component. Value of the clicked CommandBarItem will be sent to this ID.
itemValueGetter	A function that takes a CommandBarItem and returns a value to be sent to Shiny. By default it returns key of the item.

Details

- **buttonStyles** `IButtonStyles`
  Custom styles for individual button
- **cacheKey** `string`
  A custom cache key to be used for this item. If cacheKey is changed, the cache will invalidate. Defaults to key value.
- **commandBarButtonAs** `IComponentAs<ICommandBarItemProps>`
  Method to override the render of the individual command bar button. Not used when item is rendered in overflow.
- **iconOnly** `boolean`
  Show only an icon for this item, not text. Does not apply if item is in the overflow.
- **renderedInOverflow** `boolean`
  Context under which the item is being rendered. This value is mutated by the CommandBar and is useful for adjusting the onRender function.
- **tooltipHostProps** `ITooltipHostProps`
  Props for the tooltip when in iconOnly mode.
- **ariaLabel** `string`
  Accessibility text to be read by the screen reader when the user's focus enters the command bar. The screen reader will read this text after reading information about the first focusable item in the command bar.
- **buttonAs** `IComponentAs<IButtonProps>`
  Custom component for the near and far item buttons. Not used for overflow menu items.
- **className** `string`
  Additional css class to apply to the command bar
- **componentRef** `IRefObject<ICommandBar>`
  Optional callback to access the ICommandBar interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dataDidRender** `(renderedData: any) => void`
  Function to be called every time data is rendered. It provides the data that was actually rendered. A use case would be adding telemetry when a particular control is shown in an overflow or dropped as a result of onReduceData, or to count the number of renders that an implementation of onReduceData triggers.
- **farItems** `ICommandBarItemProps[]`
  Items to render on the right side (or left, in RTL). ICommandBarItemProps extends IContextualMenuItem.

- **items** `ICommandBarItemProps[]`
  Items to render. ICommandBarItemProps extends IContextualMenuItem.
- **onDataGrown** `(movedItem: ICommandBarItemProps) => void`
  Callback invoked when data has been grown.
- **onDataReduced** `(movedItem: ICommandBarItemProps) => void`
  Callback invoked when data has been reduced.
- **onGrowData** `(data: ICommandBarData) => ICommandBarData | undefined`
  Custom function to grow data if items are too small for the given space. Return `undefined` if no more steps can be taken to avoid infinate loop.
- **onReduceData** `(data: ICommandBarData) => ICommandBarData | undefined`
  Custom function to reduce data if items do not fit in given space. Return `undefined` if no more steps can be taken to avoid infinate loop.
- **overflowButtonAs** `IComponentAs<IButtonProps>`
  Custom component for the overflow button.
- **overflowButtonProps** `IButtonProps`
  Props to be passed to overflow button. If `menuProps` are passed through this prop, any items provided will be prepended to any computed overflow items.
- **overflowItems** `ICommandBarItemProps[]`
  Default items to have in the overflow menu. ICommandBarItemProps extends IContextualMenuItem.
- **shiftOnReduce** `boolean`
  When true, items will be 'shifted' off the front of the array when reduced, and unshifted during grow.
- **styles** `IStyleFunctionOrObject<ICommandBarStyleProps, ICommandBarStyles>`
  Customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

items <- function(ns) {
  list(
    CommandBarItem(
      key = ns("newItem"),
      text = "New",
      cacheKey = "myCacheKey",
      split = TRUE,
      iconProps = list(iconName = "Add"),
      subMenuProps = list(
        items = list(
```

```
            CommandBarItem(
              key = ns("emailMessage"),
              text = "Email message",
              iconProps = list(iconName = "Mail")
            ),
            CommandBarItem(
              key = ns("calendarEvent"),
              text = "Calendar event",
              iconProps = list(iconName = "Calendar")
            )
          )
        )
      ),
      CommandBarItem(
        key = ns("upload"),
        text = "Upload",
        iconProps = list(iconName = "Upload")
      ),
      CommandBarItem(
        key = ns("share"),
        text = "Share",
        iconProps = list(iconName = "Share")
      ),
      CommandBarItem(
        key = ns("download"),
        text = "Download",
        iconProps = list(iconName = "Download")
      )
    )
  }

  farItems <- function(ns) {
    list(
      CommandBarItem(
        key = ns("tile"),
        text = "Grid view",
        ariaLabel = "Grid view",
        iconOnly = TRUE,
        iconProps = list(iconName = "Tiles")
      ),
      CommandBarItem(
        key = ns("info"),
        text = "Info",
        ariaLabel = "Info",
        iconOnly = TRUE,
        iconProps = list(iconName = "Info")
      )
    )
  }

  ui <- function(id) {
    ns <- NS(id)
    tagList(
```

```
    CommandBar(
      items = items(ns),
      farItems = farItems(ns)
    ),
    textOutput(ns("commandBarItems")),
    CommandBar.shinyInput(
      inputId = ns("commandBar"),
      items = items(identity),
      farItems = farItems(identity)
    ),
    textOutput(ns("commandBar"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    commandBarItemClicked <- reactiveVal()
   observeEvent(input$newItem, commandBarItemClicked("newItem clicked (explicitly observed)"))
   observeEvent(input$upload, commandBarItemClicked("upload clicked (explicitly observed)"))
    output$commandBarItems <- renderText(commandBarItemClicked())
    output$commandBar <- renderText(input$commandBar)
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`CommandBarItem` *Command bar item*

Description

Helper function for constructing items for `CommandBar` and `CommandBar.shinyInput`.

Usage

```
CommandBarItem(
  key,
  text,
  onClick = setInputValue(inputId = key, value = 0, event = TRUE),
  ...
)
```

Arguments

`key`	Key of the item.
`text`	Text to be displayed on the menu.

onClick	A JS function that runs on item click. By default it sends input value to input[[key]]. If used within CommandBar.shinyInput, it will send the value to the input ID specified in inputId argument of CommandBar.shinyInput.
...	Additional props to pass to CommandBarItem.

Value

Item suitable for use in the CommandBar and CommandBar.shinyInput.

See Also

CommandBar

CompactPeoplePicker *PeoplePicker*

Description

The people picker (PeoplePicker) is used to select one or more entities, such as people or groups, from a list. It makes composing an email to someone, or adding them to a group, easy if you don't know their full name or email address.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
CompactPeoplePicker(...)

NormalPeoplePicker(...)

NormalPeoplePicker.shinyInput(inputId, ..., value = defaultValue)

updateNormalPeoplePicker.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **styles** `IStyleFunctionOrObject<IPeoplePickerItemSelectedStyleProps, IPeoplePickerItemSelectedStyl`
  Call to provide customized styling that will layer on top of the variant rules.
- **ValidationState** `ValidationState`
- **className** `string`
  Additional CSS class(es) to apply to the PeoplePickerItem root element.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **compact** `boolean`
  Flag that controls whether each suggested PeoplePicker item (Persona) is rendered with or without secondary text for compact look.
- **personaProps** `IPersonaProps`
  Persona props for each suggested for picking PeoplePicker item.
- **styles** `IStyleFunctionOrObject<IPeoplePickerItemSuggestionStyleProps, IPeoplePickerItemSuggestion`
  Call to provide customized styling that will layer on top of the variant rules.
- **suggestionsProps** `IBasePickerSuggestionsProps`
  General common props for all PeoplePicker items suggestions.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use the people picker to add someone to the To line of an email, or to add someone to a list.
- Use the `MemberList PeoplePicker` to display selections below the input field.

Examples

```
library(shiny)
library(shiny.fluent)

assetsUrl <- "https://static2.sharepointonline.com/files/fabric/office-ui-fabric-react-assets/"
malePersonaUrl <- paste0(assetsUrl, "persona-male.png")
femalePersonaUrl <- paste0(assetsUrl, "persona-female.png")

people <- tibble::tibble(
  key = c(1, 2, 3, 4, 5, 6, 7),
  imageUrl = c(
    femalePersonaUrl,
    malePersonaUrl,
    malePersonaUrl,
    malePersonaUrl,
    malePersonaUrl,
```

```
      femalePersonaUrl,
      malePersonaUrl
    ),
    imageInitials = c("PV", "AR", "AL", "RK", "CB", "VL", "MS"),
    text = c(
      "Annie Lindqvist",
      "Aaron Reid",
      "Alex Lundberg",
      "Roko Kolar",
      "Christian Bergqvist",
      "Valentina Lovric",
      "Maor Sharett"
    ),
    secondaryText = c(
      "Designer",
      "Designer",
      "Software Developer",
      "Financial Analyst",
      "Sr. Designer",
      "Design Developer",
      "UX Designer"
    ),
    tertiaryText = c(
      "In a meeting",
      "In a meeting",
      "In a meeting",
      "In a meeting",
      "In a meeting",
      "In a meeting",
      "In a meeting"
    ),
    optionalText = c(
      "Available at 4:00pm",
      "Available at 4:00pm",
      "Available at 4:00pm",
      "Available at 4:00pm",
      "Available at 4:00pm",
      "Available at 4:00pm",
      "Available at 4:00pm"
    ),
    isValid = c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE),
    presence = c(2, 6, 4, 1, 2, 2, 3),
    canExpand = c(NA, NA, NA, NA, NA, NA, NA)
  )

  ui <- function(id) {
    ns <- NS(id)
    tagList(
      textOutput(ns("selectedPeople")),
      NormalPeoplePicker.shinyInput(
        ns("selectedPeople"),
        options = people,
        pickerSuggestionsProps = list(
```

```
        suggestionsHeaderText = 'Matching people',
        mostRecentlyUsedHeaderText = 'Sales reps',
        noResultsFoundText = 'No results found',
        showRemoveButtons = TRUE
      )
    )
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$selectedPeople <- renderText({
      if (length(input$selectedPeople) == 0) {
        "Select recipients below:"
      } else {
        selectedPeople <- dplyr::filter(people, key %in% input$selectedPeople)
        paste("You have selected:", paste(selectedPeople$text, collapse=", "))
      }
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`ContextualMenu` *ContextualMenu*

Description

ContextualMenus are lists of commands that are based on the context of selection, mouse hover or keyboard focus. They are one of the most effective and highly used command surfaces, and can be used in a variety of places.

There are variants that originate from a command bar, or from cursor or focus. Those that come from CommandBars use a beak that is horizontally centered on the button. Ones that come from right click and menu button do not have a beak, but appear to the right and below the cursor. ContextualMenus can have submenus from commands, show selection checks, and icons.

Organize commands in groups divided by rules. This helps users remember command locations, or find less used commands based on proximity to others. One should also group sets of mutually exclusive or multiple selectable options. Use icons sparingly, for high value commands, and don't mix icons with selection checks, as it makes parsing commands difficult. Avoid submenus of submenus as they can be difficult to invoke or remember.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ContextualMenu(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the ContextualMenuItem
- **classNames** `IMenuItemClassNames`
  Classnames for different aspects of a menu item
- **componentRef** `IRefObject<IContextualMenuRenderItem>`
  Optional callback to access the IContextualMenuRenderItem interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dismissMenu** `(ev?: any, dismissAll?: boolean) => void`
  This prop will get set by ContextualMenu and can be called to close the menu this item belongs to. If dismissAll is true, all menus will be closed.
- **dismissSubMenu** `() => void`
  This prop will get set by ContextualMenu and can be called to close this item's subMenu, if present.
- **getSubmenuTarget** `() => HTMLElement | undefined`
  This prop will get set by the wrapping component and will return the element that wraps this ContextualMenuItem. Used for openSubMenu.
- **hasIcons** `boolean | undefined`
  If this item has icons
- **index** `number`
  Index of the item
- **item** `IContextualMenuItem`
  The item to display
- **onCheckmarkClick** `(item: IContextualMenuItem, ev: React.MouseEvent<HTMLElement>) => void`
  Click handler for the checkmark
- **openSubMenu** `(item: any, target: HTMLElement) => void`
  This prop will get set by ContextualMenu and can be called to open this item's subMenu, if present.
- **styles** `IStyleFunctionOrObject<IContextualMenuItemStyleProps, IContextualMenuItemStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **focusableElementIndex** `number`
- **hasCheckmarks** `boolean`

- **hasIcons** `boolean`
- **index** `number`
- **totalItemCount** `number`
- **defaultMenuItemRenderer** `(item: IContextualMenuItemRenderProps) => React.ReactNode`
- **hasCheckmarks** `boolean`
- **hasIcons** `boolean`
- **items** `IContextualMenuItem[]`
- **role** `string`
- **totalItemCount** `number`
- **alignTargetEdge** `boolean`
  If true the positioning logic will prefer to flip edges rather than to nudge the rectangle to fit within bounds, thus making sure the element aligns perfectly with target's alignment edge
- **ariaLabel** `string`
  Accessible label for the ContextualMenu's root element (inside the callout).
- **beakWidth** `number`
  The width of the beak.
- **bounds** `IRectangle | ((target?: Target, targetWindow?: Window) => IRectangle | undefined)`
  The bounding rectangle (or callback that returns a rectangle) which the contextual menu can appear in.
- **calloutProps** `ICalloutProps`
  Additional custom props for the Callout.
- **className** `string`
  Additional CSS class to apply to the ContextualMenu.
- **componentRef** `IRefObject<IContextualMenu>`
  Optional callback to access the IContextualMenu interface. Use this instead of ref for accessing the public methods and properties of the component.
- **contextualMenuItemAs** `React.ComponentClass<IContextualMenuItemProps> | React.FunctionComponent<I`
  Custom component to use for rendering individual menu items.
- **coverTarget** `boolean`
  If true, the menu will be positioned to cover the target. If false, it will be positioned next to the target.
- **delayUpdateFocusOnHover** `boolean`
  If true, the contextual menu will not be updated until focus enters the menu via other means. This will only result in different behavior when `shouldFocusOnMount = false`.

- **directionalHint** `DirectionalHint`
  How the menu should be positioned
- **directionalHintFixed** `boolean`
  If true the position will not change sides in an attempt to fit the ContextualMenu within bounds. It will still attempt to align it to whatever bounds are given.
- **directionalHintForRTL** `DirectionalHint`
  How the menu should be positioned in RTL layouts. If not specified, a mirror of `directionalHint` will be used.
- **doNotLayer** `boolean`
  If true do not render on a new layer. If false render on a new layer.
- **focusZoneProps** `IFocusZoneProps`
  Props to pass down to the FocusZone. NOTE: the default FocusZoneDirection will be used unless a direction is specified in the focusZoneProps (even if other focusZoneProps are defined)
- **gapSpace** `number`
  The gap between the ContextualMenu and the target
- **getMenuClassNames** `(theme: ITheme, className?: string) => IContextualMenuClassNames`
  Method to provide the classnames to style the contextual menu.
- **hidden** `boolean`
  If true, renders the ContextualMenu in a hidden state. Use this flag, rather than rendering a ContextualMenu conditionally based on visibility, to improve rendering performance when it becomes visible. Note: When ContextualMenu is hidden its content will not be rendered. It will only render once the ContextualMenu is visible.
- **id** `string`
  ID for the ContextualMenu's root element (inside the callout). Should be used for `aria-owns` and other such uses, rather than direct reference for programmatic purposes.
- **isBeakVisible** `boolean`
  If true then the beak is visible. If false it will not be shown.
- **isSubMenu** `boolean`
  Whether this menu is a submenu of another menu.
- **items** `IContextualMenuItem[]`
  Menu items to display.
- **labelElementId** `string`
  Used as `aria-labelledby` for the menu element inside the callout.
- **onDismiss** `(ev?: React.MouseEvent | React.KeyboardEvent, dismissAll?: boolean) => void`
  Callback when the ContextualMenu tries to close. If `dismissAll` is true then all submenus will be dismissed.
- **onItemClick** `(ev?: React.MouseEvent<HTMLElement> | React.KeyboardEvent<HTMLElement>, item?: IConte`
  Click handler which is invoked if `onClick` is not passed for individual contextual menu item. Returning true will dismiss the menu even if `ev.preventDefault()` was called.
- **onMenuDismissed** `(contextualMenu?: IContextualMenuProps) => void`
  Callback for when the menu is being closed (removing from the DOM).
- **onMenuOpened** `(contextualMenu?: IContextualMenuProps) => void`
  Callback for when the menu has been opened.

- **onRenderMenuList** `IRenderFunction<IContextualMenuListProps>`
  Method to override the render of the list of menu items.
- **onRenderSubMenu** `IRenderFunction<IContextualMenuProps>`
  Custom render function for a submenu.
- **onRestoreFocus** `(options: { originalElement?: HTMLElement | Window; containsFocus: boolean; }) => v`
  Called when the component is unmounting, and focus needs to be restored. Argument passed down contains two variables, the element that the underlying popup believes focus should go to and whether or not the popup currently contains focus. If this prop is provided, focus will not be restored automatically, you'll need to call originalElement.focus()
- **shouldFocusOnContainer** `boolean`
  Whether to focus on the contextual menu container (as opposed to the first menu item).
- **shouldFocusOnMount** `boolean`
  Whether to focus on the menu when mounted.
- **shouldUpdateWhenHidden** `boolean`
  If true, the menu will be updated even when `hidden=true`. Note that this will consume resources to update even when nothing is being shown to the user. This might be helpful if your updates are small and you want the menu to display quickly when `hidden` is set to false.
- **styles** `IStyleFunctionOrObject<IContextualMenuStyleProps, IContextualMenuStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **subMenuHoverDelay** `number`
  Delay (in milliseconds) to wait before expanding / dismissing a submenu on mouseEnter or mouseLeave
- **target** `Target`
  The target that the ContextualMenu should try to position itself based on. It can be either an element, a query selector string resolving to a valid element, or a MouseEvent. If a MouseEvent is given, the origin point of the event will be used.
- **theme** `ITheme`
  Theme provided by higher-order component.
- **title** `string`
  Title to be displayed at the top of the menu, above the items.
- **useTargetAsMinWidth** `boolean`
  If true the context menu will have a minimum width equal to the width of the target element
- **useTargetWidth** `boolean`
  If true the context menu will render as the same width as the target element

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
```

```
  div(
    DefaultButton.shinyInput(
      ns("toggleContextualMenu"),
      id = "target",
      text = "Toggle menu"
    ),
    reactOutput(ns("contextualMenu"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns

    showContextualMenu <- reactiveVal(FALSE)
    observeEvent(input$toggleContextualMenu, {
      showContextualMenu(!showContextualMenu())
    })

    output$contextualMenu <- renderReact({
      menuItems <- JS("[
        {
          key: 'newItem',
          text: 'New',
          onClick: () => console.log('New clicked'),
        },
        {
          key: 'divider_1',
          itemType: 1,
        },
        {
          key: 'rename',
          text: 'Rename',
          onClick: () => console.log('Rename clicked'),
        },
        {
          key: 'edit',
          text: 'Edit',
          onClick: () => console.log('Edit clicked'),
        },
        {
          key: 'properties',
          text: 'Properties',
          onClick: () => console.log('Properties clicked'),
        },
        {
          key: 'linkNoTarget',
          text: 'Link same window',
          href: 'http://bing.com',
        },
        {
          key: 'linkWithTarget',
          text: 'Link new window',
```

```
            href: 'http://bing.com',
            target: '_blank',
          },
          {
            key: 'linkWithOnClick',
            name: 'Link click',
            href: 'http://bing.com',
            onClick: function(){
              alert('Link clicked');
              ev.preventDefault();
            },
            target: '_blank',
          },
          {
            key: 'disabled',
            text: 'Disabled item',
            disabled: true,
            onClick: () => console.error('Disabled item should not be clickable.'),
          },
        ]")

        ContextualMenu(
          items = menuItems,
          hidden = !showContextualMenu(),
          target = "#target",
          onItemClick = JS(paste0(
            "function() {",
            "  Shiny.setInputValue('", ns("toggleContextualMenu"), "', Math.random());",
            "}"
          )),
          onDismiss = JS(paste0(
            "function() {",
            "  Shiny.setInputValue('", ns("toggleContextualMenu"), "', Math.random());",
            "}"
          ))
        )
      })
    })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

DatePicker *DatePicker*

Description

Picking a date can be tough without context. A date picker (`DatePicker`) offers a drop-down control that's optimized for picking a single date from a calendar view where contextual information like

the day of the week or fullness of the calendar is important. You can modify the calendar to provide additional context or to limit available dates.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
DatePicker(...)

DatePicker.shinyInput(inputId, ..., value = defaultValue)

updateDatePicker.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **allFocusable** `boolean`
  Allows all elements to be focused, including disabled ones
- **allowTextInput** `boolean`
  Whether the DatePicker allows input a date string directly or not
- **ariaLabel** `string`
  Aria Label for TextField of the DatePicker for screen reader users.
- **borderless** `boolean`
  Determines if DatePicker has a border.
- **calendarAs** `IComponentAs<ICalendarProps>`
  Custom Calendar to be used for date picking
- **calendarProps** `ICalendarProps`
  Pass calendar props to calendar component
- **calloutProps** `ICalloutProps`
  Pass callout props to callout component
- **className** `string`
  Optional Classname for datepicker root element .

- **componentRef** `IRefObject<IDatePicker>`
  Optional callback to access the IDatePicker interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dateTimeFormatter** `ICalendarFormatDateCallbacks`
  Apply additional formating to dates, for example localized date formatting.
- **disableAutoFocus** `boolean`
  Whether the DatePicker should open automatically when the control is focused
- **disabled** `boolean`
  Disabled state of the DatePicker.
- **firstDayOfWeek** `DayOfWeek`
  The first day of the week for your locale.
- **firstWeekOfYear** `FirstWeekOfYear`
  Defines when the first week of the year should start, FirstWeekOfYear.FirstDay, FirstWeekOfYear.FirstFullWeek or FirstWeekOfYear.FirstFourDayWeek are the possible values
- **formatDate** `(date?: Date) => string`
  Optional method to format the chosen date to a string to display in the DatePicker
- **highlightCurrentMonth** `boolean`
  Whether the month picker should highlight the current month
- **highlightSelectedMonth** `boolean`
  Whether the month picker should highlight the selected month
- **initialPickerDate** `Date`
  The initially highlighted date in the calendar picker
- **isMonthPickerVisible** `boolean`
  Whether the month picker is shown beside the day picker or hidden.
- **isRequired** `boolean`
  Whether the DatePicker is a required field or not
- **label** `string`
  Label for the DatePicker
- **maxDate** `Date`
  The maximum allowable date.
- **minDate** `Date`
  The minimum allowable date.
- **onAfterMenuDismiss** `() => void`
  Callback that runs after DatePicker's menu (Calendar) is closed
- **onSelectDate** `(date: Date | null | undefined) => void`
  Callback issued when a date is selected
- **parseDateFromString** `(dateStr: string) => Date | null`
  Optional method to parse the text input value to date, it is only useful when allowTextInput is set to true
- **pickerAriaLabel** `string`
  Aria label for date picker popup for screen reader users.
- **placeholder** `string`
  Placeholder text for the DatePicker

- **showCloseButton** `boolean`
  Whether the CalendarDay close button should be shown or not.
- **showGoToToday** `boolean`
  Whether the "Go to today" link should be shown or not
- **showMonthPickerAsOverlay** `boolean`
  Show month picker on top of date picker when visible.
- **showWeekNumbers** `boolean`
  Whether the calendar should show the week number (weeks 1 to 53) before each week row
- **strings** `IDatePickerStrings`
  Localized strings to use in the DatePicker
- **styles** `IStyleFunctionOrObject<IDatePickerStyleProps, IDatePickerStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **tabIndex** `number`
  The tabIndex of the TextField
- **textField** `ITextFieldProps`
  Pass textField props to textField component. Prop name is "textField" for compatiblity with upcoming slots work.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **today** `Date`
  Value of today. If null, current time in client machine will be used.
- **underlined** `boolean`
  Whether or not the Textfield of the DatePicker is underlined.
- **value** `Date`
  Default value of the DatePicker, if any

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use the control the way it's designed and built. Don't break it apart.

Content:

- The control provides the date in a specific format. If the date can be entered in an input field, provide helper text in the appropriate format.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)
```

```
ui <- function(id) {
  ns <- NS(id)
  div(
    DatePicker.shinyInput(ns("date"), value = "2020-06-25T22:00:00.000Z"),
    textOutput(ns("dateValue")),
    h3("If `value` is missing, default to system date"),
    DatePicker.shinyInput(ns("date2")),
    textOutput(ns("dateDefault")),
    h3("If `value` is NULL, return NULL"),
    DatePicker.shinyInput(ns("date3"), value = NULL, placeholder = "I am placeholder!"),
    textOutput(ns("dateNull"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$dateValue <- renderText({
      sprintf("Value: %s", input$date)
    })
    output$dateDefault <- renderText({
      sprintf("Value: %s", input$date2)
    })
    output$dateNull <- renderText({
      sprintf("Value: %s", deparse(input$date3))
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Supplying custom strings for DatePicker
ui <- function(id) {
  fluentPage(
    DatePicker.shinyInput(
      "date",
      value = Sys.Date(),
      strings = list(
        months = list(
          "January", "February", "March", "April",
          "May", "June", "July", "August",
          "September", "October", "November", "December"
        ),
        shortMonths = list(
          "Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        ),
        days = list(
```

```
            "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"
          ),
          shortDays = list("S", "M", "T", "W", "T", "F", "S"),
          goToToday = "Go to today",
          prevMonthAriaLabel = "Go to previous month",
          nextMonthAriaLabel = "Go to next month",
          prevYearAriaLabel = "Go to previous year",
          nextYearAriaLabel = "Go to next year"
        )
      )
    )
  }

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

  if (interactive()) {
    shinyApp(ui("app"), function(input, output) server("app"))
  }
```

`DetailsList` *DetailsList*

Description

A details list (`DetailsList`) is a robust way to display an information-rich collection of items, and allow people to sort, group, and filter the content. Use a details list when information density is critical.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
DetailsList(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **onColumnDragEnd** `(props: { dropLocation?: ColumnDragEndLocation; }, event: MouseEvent) => void` Callback to notify the column dragEnd event to List Need this to check whether the dragEnd has happened on corresponding list or outside of the list

- **cellStyleProps** `ICellStyleProps`
  Custom styles for cell rendering.
- **column** `IColumn`
  The column definition for the component instance.
- **columnIndex** `number`
  The column index for the component instance.
- **componentRef** `() => void`
  A reference to the component instance.
- **dragDropHelper** `IDragDropHelper | null`
  The drag and drop helper for the component instance.
- **isDraggable** `boolean`
  Whether or not the column can be re-ordered via drag and drop.
- **isDropped** `boolean`
  Whether or not the column has been dropped via drag and drop.
- **onColumnClick** `(ev: React.MouseEvent<HTMLElement>, column: IColumn) => void`
  Callback fired when click event occurs.
- **onColumnContextMenu** `(column: IColumn, ev: React.MouseEvent<HTMLElement>) => void`
  Callback fired on contextual menu event to provide contextual menu UI.
- **onRenderColumnHeaderTooltip** `IRenderFunction<IDetailsColumnRenderTooltipProps>`
  Render function for providing a column header tooltip.
- **parentId** `string`
  Parent ID used for accessibility label(s).
- **setDraggedItemIndex** `(itemIndex: number) => void`

- **styles** `IStyleFunctionOrObject<IDetailsColumnStyleProps, IDetailsColumnStyles>`
  The component styles to respect during render.
- **theme** `ITheme`
  The theme object to respect during render.
- **updateDragInfo** `(props: { itemIndex: number; }, event?: MouseEvent) => void`
  Callback on drag and drop event.
- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **columns** `IColumn[]`
  Column metadata
- **selection** `ISelection`
  Selection from utilities
- **selectionMode** `SelectionMode`
  Selection mode
- **onRenderFooter** `IRenderFunction<IDetailsGroupDividerProps>`

- **onRenderHeader** `IRenderFunction<IDetailsGroupDividerProps>`

- **ariaLabel** `string`
  ariaLabel for the entire header
- **ariaLabelForSelectAllCheckbox** `string`
  ariaLabel for the header checkbox that selects or deselects everything
- **ariaLabelForSelectionColumn** `string`
  ariaLabel for the selection column
- **ariaLabelForToggleAllGroupsButton** `string`
  ariaLabel for expand/collapse group button
- **className** `string`
  Overriding class name
- **collapseAllVisibility** `CollapseAllVisibility`
  Whether to collapse for all visibility
- **columnReorderOptions** `IColumnReorderOptions`
  Column reordering options
- **columnReorderProps** `IColumnReorderHeaderProps`
  Column reordering options
- **componentRef** `IRefObject<IDetailsHeader>`
  Ref to the component itself
- **isAllCollapsed** `boolean`
  Whether or not all is collapsed
- **layoutMode** `DetailsListLayoutMode`
  Layout mode - fixedColumns or justified
- **minimumPixelsForDrag** `number`
  Minimum pixels to be moved before dragging is registered
- **onColumnAutoResized** `(column: IColumn, columnIndex: number) => void`
  Callback for when column is automatically resized
- **onColumnClick** `(ev: React.MouseEvent<HTMLElement>, column: IColumn) => void`
  Callback for when the column is clicked
- **onColumnContextMenu** `(column: IColumn, ev: React.MouseEvent<HTMLElement>) => void`
  Callback for when the column needs to show a context menu
- **onColumnIsSizingChanged** `(column: IColumn, isSizing: boolean) => void`
  Callback for when column sizing has changed
- **onColumnResized** `(column: IColumn, newWidth: number, columnIndex: number) => void`
  Callback for when column is resized
- **onRenderColumnHeaderTooltip** `IRenderFunction<IDetailsColumnRenderTooltipProps>`
  Callback to render a tooltip for the column header
- **onRenderDetailsCheckbox** `IRenderFunction<IDetailsCheckboxProps>`
  If provided, can be used to render a custom checkbox
- **onToggleCollapseAll** `(isAllCollapsed: boolean) => void`
  Callback for when collapse all is toggled
- **selectAllVisibility** `SelectAllVisibility`
  Select all button visibility

- **styles** `IStyleFunctionOrObject<IDetailsHeaderStyleProps, IDetailsHeaderStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme from the Higher Order Component
- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **columns** `IColumn[]`
  Column metadata
- **selection** `ISelection`
  Selection from utilities
- **selectionMode** `SelectionMode`
  Selection mode
- **cellStyleProps** `ICellStyleProps`
  Rules for rendering column cells.
- **checkboxVisibility** `CheckboxVisibility | undefined`
  Checkbox visibility
- **columns** `IColumn[]`
  Column metadata
- **groupNestingDepth** `number`
  Nesting depth of a grouping
- **indentWidth** `number | undefined`
  How much to indent
- **rowWidth** `number`
  Minimum width of the row.
- **selection** `ISelection | undefined`
  Selection from utilities
- **selectionMode** `SelectionMode | undefined`
  Selection mode
- **viewport** `IViewport | undefined`
  View port of the virtualized list
- **ariaLabel** `string`
  Accessible label describing or summarizing the list.
- **ariaLabelForGrid** `string`
  Accessible label for the grid within the list.
- **ariaLabelForListHeader** `string`
  Accessible label for the list header.
- **ariaLabelForSelectAllCheckbox** `string`
  Accessible label for the select all checkbox.
- **ariaLabelForSelectionColumn** `string`
  Accessible label for the name of the selection column.

- **cellStyleProps** `ICellStyleProps`
  Props impacting the render style of cells. Since these have an impact on calculated column widths, they are handled separately from normal theme styling, but they are passed to the styling system.
- **checkboxCellClassName** `string`
  Class name to add to the cell of a checkbox.
- **checkboxVisibility** `CheckboxVisibility`
  Controls the visibility of selection check box.
- **checkButtonAriaLabel** `string`
  Accessible label for the check button.
- **className** `string`
  Class name to add to the root element.
- **columnReorderOptions** `IColumnReorderOptions`
  Options for column reordering using drag and drop.
- **columns** `IColumn[]`
  column defitions. If none are provided, default columns will be created based on the items' properties.
- **compact** `boolean`
  Whether to render in compact mode.
- **componentRef** `IRefObject<IDetailsList>`
  Callback to access the IDetailsList interface. Use this instead of ref for accessing the public methods and properties of the component.
- **constrainMode** `ConstrainMode`
  Controls how the list contrains overflow.
- **disableSelectionZone** `boolean`
  Whether to disable the built-in SelectionZone, so the host component can provide its own.
- **dragDropEvents** `IDragDropEvents`
  Map of callback functions related to row drag and drop functionality.
- **enableUpdateAnimations** `boolean`
  Whether to animate updates
- **enterModalSelectionOnTouch** `boolean`
  Whether the selection zone should enter modal state on touch.
- **getCellValueKey** `(item?: any, index?: number, column?: IColumn) => string`
  If provided, will be the "default" item column cell value return. A column's `getValueKey` can override `getCellValueKey`.
- **getGroupHeight** `IGroupedListProps['getGroupHeight']`
  Callback to override default group height calculation used by list virtualization.
- **getKey** `(item: any, index?: number) => string`
  Callback to get the item key, to be used in the selection and on render. Must be provided if sorting or filtering is enabled.
- **getRowAriaDescribedBy** `(item: any) => string`
  Callback to get the aria-describedby IDs (space-separated strings) of elements that describe the item.

- **getRowAriaLabel** `(item: any) => string`
  Callback to get the aria-label string for a given item.
- **groupProps** `IDetailsGroupRenderProps`
  Override properties to render groups.
- **groups** `IGroup[]`
  Grouping instructions.
- **indentWidth** `number`
  Override for the indent width used for group nesting.
- **initialFocusedIndex** `number`
  Default index to set focus to once the items have rendered and the index exists.
- **isHeaderVisible** `boolean`
  Controls the visibility of the header.
- **isPlaceholderData** `boolean`
  Set this to true to indicate that the items being displayed are placeholder data.
- **items** `any[]`
  The items to render.
- **layoutMode** `DetailsListLayoutMode`
  Controls how the columns are adjusted.
- **listProps** `IListProps`
  Properties to pass through to the List components being rendered.
- **minimumPixelsForDrag** `number`
  The minimum mouse move distance to interpret the action as drag event.
- **onActiveItemChanged** `(item?: any, index?: number, ev?: React.FocusEvent<HTMLElement>) => void`
  Callback for when an item in the list becomes active by clicking anywhere inside the row or navigating to it with the keyboard.
- **onColumnHeaderClick** `(ev?: React.MouseEvent<HTMLElement>, column?: IColumn) => void`
  Callback for when the user clicks on the column header.
- **onColumnHeaderContextMenu** `(column?: IColumn, ev?: React.MouseEvent<HTMLElement>) => void`
  Callback for when the user asks for a contextual menu (usually via right click) from a column header.
- **onColumnResize** `(column?: IColumn, newWidth?: number, columnIndex?: number) => void`
  Callback fired on column resize
- **onDidUpdate** `(detailsList?: DetailsListBase) => void`
  Callback for when the list has been updated. Useful for telemetry tracking externally.
- **onItemContextMenu** `(item?: any, index?: number, ev?: Event) => void | boolean`
  Callback for when the context menu of an item has been accessed. If undefined or false is returned, `ev.preventDefault()` will be called.
- **onItemInvoked** `(item?: any, index?: number, ev?: Event) => void`
  Callback for when a given row has been invoked (by pressing enter while it is selected.)
- **onRenderCheckbox** `IRenderFunction<IDetailsListCheckboxProps>`
  If provided, can be used to render a custom checkbox.
- **onRenderDetailsFooter** `IRenderFunction<IDetailsFooterProps>`
  An override to render the details footer.

- **onRenderDetailsHeader** `IRenderFunction<IDetailsHeaderProps>`
  An override to render the details header.
- **onRenderItemColumn** `(item?: any, index?: number, column?: IColumn) => React.ReactNode`
  If provided, will be the "default" item column renderer method. This affects cells within the rows, not the rows themselves. If a column definition provides its own `onRender` method, that will be used instead of this.
- **onRenderMissingItem** `(index?: number, rowProps?: IDetailsRowProps) => React.ReactNode`
  Callback for what to render when the item is missing.
- **onRenderRow** `IRenderFunction<IDetailsRowProps>`
  Callback to override the default row rendering.
- **onRowDidMount** `(item?: any, index?: number) => void`
  Callback for when a given row has been mounted. Useful for identifying when a row has been rendered on the page.
- **onRowWillUnmount** `(item?: any, index?: number) => void`
  Callback for when a given row has been unmounted. Useful for identifying when a row has been removed from the page.
- **onShouldVirtualize** `(props: IListProps) => boolean`
  Callback to determine whether the list should be rendered in full, or virtualized.

Virtualization will add and remove pages of items as the user scrolls them into the visible range. This benefits larger list scenarios by reducing the DOM on the screen, but can negatively affect performance for smaller lists.

The default implementation will virtualize when this callback is not provided.

- **rowElementEventMap** `{ eventName: string; callback: (context: IDragDropContext, event?: any) => vo`
  Event names and corresponding callbacks that will be registered to rendered row elements.
- **selection** `ISelection`
  Selection model to track selection state.
- **selectionMode** `SelectionMode`
  Controls how/if the details list manages selection. Options include none, single, multiple
- **selectionPreservedOnEmptyClick** `boolean`
  By default, selection is cleared when clicking on an empty (non-focusable) section of the screen. Setting this value to true overrides that behavior and maintains selection.
- **selectionZoneProps** `ISelectionZoneProps`
  Additional props to pass through to the SelectionZone created by default.
- **setKey** `string`
  A key that uniquely identifies the given items. If provided, the selection will be reset when the key changes.
- **shouldApplyApplicationRole** `boolean`
  Whether the role `application` should be applied to the list.
- **styles** `IStyleFunctionOrObject<IDetailsListStyleProps, IDetailsListStyles>`
  Custom overrides to the themed or default styles.
- **theme** `ITheme`
  Theme provided by a higher-order component.

- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **usePageCache** `boolean`
  Whether to enable render page caching. This is an experimental performance optimization that is off by default.
- **useReducedRowRenderer** `boolean`
  Whether to re-render a row only when props changed. Might cause regression when depending on external updates.
- **viewport** `IViewport`
  Viewport info, provided by the `withViewport` decorator.
- **cellsByColumn** `{ [columnKey: string]: React.ReactNode; }`
  Optional pre-rendered content per column. Preferred over onRender or onRenderItemColumn if provided.
- **checkboxCellClassName** `string`
  Class name for the checkbox cell
- **checkButtonAriaLabel** `string`
  Check button's aria label
- **className** `string`
  Overriding class name
- **collapseAllVisibility** `CollapseAllVisibility`
  Collapse all visibility
- **compact** `boolean`
  Whether to render in compact mode
- **componentRef** `IRefObject<IDetailsRow>`
  Ref of the component
- **dragDropEvents** `IDragDropEvents`
  Handling drag and drop events
- **dragDropHelper** `IDragDropHelper`
  Helper for the drag and drop
- **enableUpdateAnimations** `boolean`
  Whether to animate updates
- **eventsToRegister** `{ eventName: string; callback: (item?: any, index?: number, event?: any) => void; }`
  A list of events to register
- **getRowAriaDescribedBy** `(item: any) => string`
  Callback for getting the row aria-describedby
- **getRowAriaLabel** `(item: any) => string`
  Callback for getting the row aria label
- **item** `any`
  Data source for this component
- **itemIndex** `number`
  Index of the collection of items of the DetailsList

- **onDidMount** `(row?: DetailsRowBase) => void`
  Callback for did mount for parent
- **onRenderCheck** `(props: IDetailsRowCheckProps) => JSX.Element`
  Callback for rendering a checkbox
- **onRenderDetailsCheckbox** `IRenderFunction<IDetailsCheckboxProps>`
  If provided, can be used to render a custom checkbox
- **onWillUnmount** `(row?: DetailsRowBase) => void`
  Callback for will mount for parent
- **rowFieldsAs** `React.ComponentType<IDetailsRowFieldsProps>`
  DOM element into which to render row field
- **styles** `IStyleFunctionOrObject<IDetailsRowStyleProps, IDetailsRowStyles>`
  Overriding styles to this row
- **theme** `ITheme`
  Theme provided by styled() function
- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **useReducedRowRenderer** `boolean`
  Rerender DetailsRow only when props changed. Might cause regression when depending on external updates.
- **anySelected** `boolean`
  Is any selected - also true for isSelectionModal
- **canSelect** `boolean`
  Can this checkbox be selectable
- **checkClassName** `string`
  The classname to be passed down to Check component
- **className** `string`
  Optional className to attach to the slider root element.
- **compact** `boolean`
  Is this in compact mode?
- **isHeader** `boolean`
  Is the check part of the header in a DetailsList
- **isVisible** `boolean`
  Whether or not this checkbox is visible
- **onRenderDetailsCheckbox** `IRenderFunction<IDetailsCheckboxProps>`
  If provided, can be used to render a custom checkbox
- **selected** `boolean`
  Whether or not this check is selected
- **styles** `IStyleFunctionOrObject<IDetailsRowCheckStyleProps, IDetailsRowCheckStyles>`
  Style override
- **theme** `ITheme`
  Theme provided by High-Order Component.

- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **cellStyleProps** `ICellStyleProps`
  Style properties to customize cell render output.
- **columns** `IColumn[]`
  Columns metadata
- **columnStartIndex** `number`
  Index to start for the column
- **compact** `boolean`
  whether to render as a compact field
- **enableUpdateAnimations** `boolean`
- **item** `any`
  Data source for this component
- **itemIndex** `number`
  The item index of the collection for the DetailsList
- **rowClassNames** `{ [k in keyof Pick<IDetailsRowStyles, 'isMultiline' | 'isRowHeader' | 'cell' | 'cell`
  Subset of classnames currently generated in DetailsRow that are used within DetailsRow-Fields.
- **columns** `IColumn[]`
  Column metadata
- **selection** `ISelection`
  Selection from utilities
- **selectionMode** `SelectionMode`
  Selection mode
- **ariaLabelForShimmer** `string`
  Aria label for shimmer. Set on grid while shimmer is enabled.
- **detailsListStyles** `IDetailsListProps['styles']`
  DetailsList styles to pass through.
- **enableShimmer** `boolean`
  Boolean flag to control when to render placeholders vs real items. It's up to the consumer app to know when fetching of the data is done to toggle this prop.
- **onRenderCustomPlaceholder** `(rowProps: IDetailsRowProps, index?: number, defaultRender?: (props: ID`
  Custom placeholder renderer to be used when in need to override the default placeholder of a DetailsRow. `rowProps` argument is passed to leverage the calculated column measurements done by DetailsList or you can use the optional arguments of item `index` and `defaultRender` to execute additional logic before rendering the default placeholder.
- **removeFadingOverlay** `boolean`
  Determines whether to remove a fading out to bottom overlay over the shimmering items used to further emphasize the unknown number of items that will be fetched.
- **shimmerLines** `number`
  Number of shimmer placeholder lines to render.

- **shimmerOverlayStyles** `IStyleFunctionOrObject<IShimmeredDetailsListStyleProps, IShimmeredDetailsLi`
  Custom styles to override the styles specific to the ShimmeredDetailsList root area.
- **styles** `IStyleFunctionOrObject<IShimmeredDetailsListStyleProps, IShimmeredDetailsListStyles>`
  Custom styles to override the styles specific to the ShimmeredDetailsList root area.
- **skipViewportMeasures** `boolean`
  Whether or not to use ResizeObserver (if available) to detect and measure viewport on 'resize' events.

Falls back to window 'resize' event.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- List items are composed of selection, icon, and name columns at minimum. You can include other columns, such as date modified, or any other metadata field associated with the collection.
- Avoid using file type icon overlays to denote status of a file as it can make the entire icon unclear.
- If there are multiple lines of text in a column, consider the variable row height variant.
- Give columns ample default width to display information.

Content:

- Use sentence-style capitalization for column headers—only capitalize the first word. For more info, see `[Capitalization]` in the Microsoft Writing Style Guide.

`[capitalization]`: https://docs.microsoft.com/style-guide/capitalization

FAQ:

My scrollable content isn't updating on scroll. What should I do?:
Add the `data-is-scrollable="true"` attribute to your scrollable element containing the DetailsList.
By default, the List used within DetailsList will use the `body` element as the scrollable element. If you contain the List within a scrollable `div` using `overflow: auto` or `scroll`, the List needs to listen for scroll events on that element instead. On initialization, the List will traverse up the DOM looking for the first element with the `data-is-scrollable` attribute to know which element to listen to for knowing when to re-evaulate the visible window.

My List is not re-rendering when I mutate its items. What should I do?:
To determine if the List within DetailsList should re-render its contents, the component performs a referential equality check within its `shouldComponentUpdate` method. This is done to minimize the performance overhead associated with re-rendering the virtualized List pages, as recommended by the React documentation.
As a result of this implementation, the inner List will not determine it should re-render if the array values are mutated. To avoid this problem, we recommend re-creating the items array backing the DetailsList by using a method such as `Array.prototype.concat` or ES6 spread syntax shown below:

```
public appendItems(): void {
  const { items } = this.state;

  this.setState({
    items: [...items, ...['Foo', 'Bar']]
  })
}

public render(): JSX.Element {
  const { items } = this.state;

  return <DetailsList items={items} />;
}
```

By re-creating the items array without mutating the values, the inner List will correctly determine its contents have changed and it should then re-render with the new values.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

items <- list(
  list(key = "1", name = "Mark", surname = "Swanson"),
  list(key = "2", name = "Josh", surname = "Johnson")
)

columns <- list(
  list(key = "name", fieldName = "name", name = "Name"),
  list(key = "surname", fieldName = "surname", name = "Surname")
)


ui <- function(id) {
  ns <- NS(id)
  DetailsList(items = items, columns = columns)
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Custom columns text alignment and formatting
```

```
items <- list(
  list(
    key = "1",
    name = "Mark",
    number = "2"
  ),
  list(
    key = "2",
    name = "Josh",
    number = "1"
  )
)

columns <- list(
  list(
    key = "name",
    fieldName = "name",
    name = "Name"
  ),
  list(
    key = "number",
    fieldName = "number",
    name = "Number"
  )
)

ui <- function(id) {
  DetailsList(
    items = items,
    columns = columns,
    onRenderItemColumn = JS("(item, index, column) => {
      const fieldContent = item[column.fieldName]
      switch (column.key) {
        case 'name':
          return React.createElement(
            'span',
            {
              style: { textAlign: 'right', width: '100%', display: 'block' }
            },
            fieldContent
          );
        case 'number':
          return React.createElement(
            'span',
            {
              style: { textAlign: 'left', width: '100%', display: 'block' }
            },
            `%${fieldContent}`
          );
        default:
          return React.createElement('span', null, fieldContent);
      }
    }")
```

```
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 3
library(shiny)
library(shiny.fluent)

# Selecting rows in DetailsList
CustomComponents <- tags$script(HTML("(function() {
  const React = jsmodule['react'];
  const Fluent = jsmodule['@fluentui/react'];
  const Shiny = jsmodule['@/shiny'];
  const CustomComponents = jsmodule['CustomComponents'] ??= {};

  function useSelection(inputId) {
    const selection = React.useRef(new Fluent.Selection({
      onSelectionChanged() {
       const value = this.getSelectedIndices().map(i => i + 1); // R uses 1-based indexing.
        Shiny.setInputValue(inputId, value);
       }
    }));
    return selection.current;
  }

  CustomComponents.DetailsList = function DetailsList({ inputId, ...rest }) {
    const selection = useSelection(inputId);
    return React.createElement(Fluent.DetailsList, { selection, ...rest });
  }
})()"))

DetailsList.shinyInput <- function(inputId, ...) {
  shiny.react::reactElement(
    module = "CustomComponents",
    name = "DetailsList",
    props = shiny.react::asProps(inputId = inputId, ...),
    deps = shinyFluentDependency()
  )
}

items <- list(
  list(name = "Apple"),
  list(name = "Banana"),
  list(name = "Cherry")
)
```

```
ui <- function(id) {
  ns <- NS(id)
  tagList(
    CustomComponents,
    DetailsList.shinyInput(ns("selection"), items = items),
    textOutput(ns("text"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$text <- renderText(paste(input$selection, collapse = ", "))
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Dialog` *Dialog*

Description

A dialog box (`Dialog`) is a temporary pop-up that takes focus from the page or app and requires people to interact with it. It's primarily used for confirming actions, such as deleting a file, or asking people to make a choice.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Dialog(...)

DialogFooter(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Optional override class name
- **closeButtonAriaLabel** `string`
  Label to be passed to to aria-label of close button

- **componentRef** `IRefObject<IDialogContent>`
  Optional callback to access the IDialogContent interface. Use this instead of ref for accessing the public methods and properties of the component.
- **draggableHeaderClassName** `string`
  The classname for when the header is draggable
- **isMultiline** `boolean`
  Is inside a multiline wrapper
- **onDismiss** `(ev?: React.MouseEvent<HTMLButtonElement>) => any`
  Callback for when the Dialog is dismissed from the close button or light dismiss, before the animation completes.
- **responsiveMode** `ResponsiveMode`
  Responsive mode passed in from decorator.
- **showCloseButton** `boolean`
  Show an 'x' close button in the upper-right corner
- **styles** `IStyleFunctionOrObject<IDialogContentStyleProps, IDialogContentStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **subText** `string`
  The subtext to display in the dialog
- **subTextId** `string`
  The Id for subText container
- **theme** `ITheme`
  Theme provided by HOC.
- **title** `string | JSX.Element`
  The title text to display at the top of the dialog.
- **titleId** `string`
  The Id for title container
- **titleProps** `React.HTMLAttributes<HTMLDivElement>`
  The props for title container.
- **topButtonsProps** `IButtonProps[]`
  Other top buttons that will show up next to the close button
- **type** `DialogType`
  The type of Dialog to display.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDialogFooter>`
  Gets the component ref.
- **styles** `IStyleFunctionOrObject<IDialogFooterStyleProps, IDialogFooterStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **ariaDescribedById** `string`
  Optional id for aria-DescribedBy

- **ariaLabelledById** `string`
  Optional id for aria-LabelledBy
- **className** `string`
  Optional class name to be added to the root class
- **componentRef** `IRefObject<IDialog>`
  Optional callback to access the IDialog interface. Use this instead of ref for accessing the public methods and properties of the component.
- **containerClassName** `string`
  Optional override for container class
- **contentClassName** `string`
  Optional override content class
- **dialogContentProps** `IDialogContentProps`
  Props to be passed through to Dialog Content
- **hidden** `boolean`
  Whether the dialog is hidden.
- **isBlocking** `boolean`
  Whether the dialog can be light dismissed by clicking outside the dialog (on the overlay).
- **isDarkOverlay** `boolean`
  Whether the overlay is dark themed.
- **isOpen** `boolean`
  Whether the dialog is displayed. Deprecated, use `hidden` instead.
- **maxWidth** `ICSSRule | ICSSPixelUnitRule`
  Sets the maximum width for the dialog. It limits the width property to be larger than the value specified in max-width.
- **minWidth** `ICSSRule | ICSSPixelUnitRule`
  Sets the minimum width of the dialog. It limits the width property to be not smaller than the value specified in min-width.
- **modalProps** `IModalProps`
  Props to be passed through to Modal
- **onDismiss** `(ev?: React.MouseEvent<HTMLButtonElement>) => any`
  A callback function for when the Dialog is dismissed from the close button or light dismiss. Can also be specified separately in content and modal.
- **onDismissed** `() => any`
  A callback function which is called after the Dialog is dismissed and the animation is complete.
- **onLayerDidMount** `() => void`
  A callback function for when the Dialog content is mounted on the overlay layer
- **onLayerMounted** `() => void`
  Deprecated at 0.81.2, use `onLayerDidMount` instead.
- **styles** `IStyleFunctionOrObject<IDialogStyleProps, IDialogStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **subText** `string`
  The subtext to display in the dialog.

- **theme** `ITheme`
  Theme provided by HOC.
- **title** `string | JSX.Element`
  The title text to display at the top of the dialog.
- **topButtonsProps** `IButtonProps[]`
  Other top buttons that will show up next to the close button
- **type** `DialogType`
  The type of Dialog to display.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Don't use more than three buttons.
- Dialog boxes consist of a header, body, and footer.
- Validate that people's entries are acceptable before closing the dialog box. Show an inline validation error near the field they must correct.
- Blocking dialogs should be used very sparingly, only when it is critical that people make a choice or provide information before they can proceed. Blocking dialogs are generally used for irreversible or potentially destructive tasks. They're typically paired with an overlay without a light dismiss.

Header:

- Locks to the top of the dialog.
- May include an icon to the left of the title.
- Includes a Close button in the top-right corner.

Footer:

- Lock buttons to the bottom of the dialog.
- Includes one primary button. A secondary button is optional.

Width:

- Maximum is 340 pixels.
- Minimum is 288 pixels.

Height:

- Maximum is 340 pixels.
- Minimum is 172 pixels.

Content:

Title:

- Keep the title as concise as possible.
- Don't use periods at the end of titles.
- This mandatory content should explain the main information in a clear, concise, and specific statement or question. For example, "Delete this file?" instead of "Are you sure?"

- The title shouldn't be a description of the body content. For example, don't use "Error" as a title. Instead, use an informative statement like "Your session ended."
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

Body copy (Optional):

- Don't restate the title in the body.
- Use ending punctuation on sentences.
- Use actionable language, with the most important information at the beginning.
- Use the optional body content area for additional info or instructions, if needed. Only include information needed to help people make a decision.

Button labels:

- Write button labels that are specific responses to the main information in the title. The title "Delete this file?" could have buttons labeled "Delete" and "Cancel".
- Be concise. Limit labels to one or two words. Usually a single verb is best. Include a noun if there is any room for interpretation about what the verb means. For example, "Delete" or "Delete file".

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    DefaultButton.shinyInput(ns("showDialog"), text = "Open dialog"),
    reactOutput(ns("reactDialog"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns

    isDialogOpen <- reactiveVal(FALSE)
    output$reactDialog <- renderReact({
      dialogContentProps <- list(
        type = 0,
        title = "Missing Subject",
        closeButtonAriaLabel = "Close",
        subText = "Do you want to send this message without a subject?"
      )
      Dialog(
        hidden = !isDialogOpen(),
        onDismiss = JS(paste0(
          "function() {",
          "  Shiny.setInputValue('", ns("hideDialog"),"', Math.random());",
          "}"
        )),
        dialogContentProps = dialogContentProps,
```

```
        modalProps = list(),
        DialogFooter(
          PrimaryButton.shinyInput(ns("dialogSend"), text = "Send"),
          DefaultButton.shinyInput(ns("dialogDontSend"), text = "Don't send")
        )
      )
    })

    observeEvent(input$showDialog, isDialogOpen(TRUE))
    observeEvent(input$hideDialog, isDialogOpen(FALSE))
    observeEvent(input$dialogSend, isDialogOpen(FALSE))
    observeEvent(input$dialogDontSend, isDialogOpen(FALSE))
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`DocumentCard` *DocumentCard*

Description

A document card (`DocumentCard`) represents a file, and contains additional metadata or actions. This offers people a richer view into a file than the typical grid view.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
DocumentCard(...)

DocumentCardActions(...)

DocumentCardActivity(...)

DocumentCardDetails(...)

DocumentCardImage(...)

DocumentCardLocation(...)

DocumentCardLogo(...)

DocumentCardPreview(...)
```

```
DocumentCardStatus(...)

DocumentCardTitle(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **actions** `IButtonProps[]`
  The actions available for this document.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardActions>`
  Gets the component ref.
- **styles** `IStyleFunctionOrObject<IDocumentCardActionsStyleProps, IDocumentCardActionsStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **views** `Number`
  The number of views this document has received.
- **activity** `string`
  Describes the activity that has taken place, such as "Created Feb 23, 2016".
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardActivity>`
  Gets the component ref.
- **people** `IDocumentCardActivityPerson[]`
  One or more people who are involved in this activity.
- **styles** `IStyleFunctionOrObject<IDocumentCardActivityStyleProps, IDocumentCardActivityStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardDetails>`
  Gets the component ref.
- **styles** `IStyleFunctionOrObject<IDocumentCardDetailsStyleProps, IDocumentCardDetailsStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardImage>`
  Gets the component ref.
- **height** `number`
  If provided, forces the preview image to be this height.
- **iconProps** `IIconProps`
  The props for the icon associated with this document type.
- **imageFit** `ImageFit`
  Used to determine how to size the image to fit the dimensions of the component. If both dimensions are provided, then the image is fit using ImageFit.scale, otherwise ImageFit.none is used.
- **imageSrc** `string`
  Path to the preview image.
- **styles** `IStyleFunctionOrObject<IDocumentCardImageStyleProps, IDocumentCardImageStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **width** `number`
  If provided, forces the preview image to be this width.
- **ariaLabel** `string`
  Aria label for the link to the document location.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardLocation>`
  Gets the component ref.
- **location** `string`
  Text for the location of the document.
- **locationHref** `string`
  URL to navigate to for this location.
- **onClick** `(ev?: React.MouseEvent<HTMLElement>) => void`
  Function to call when the location is clicked.
- **styles** `IStyleFunctionOrObject<IDocumentCardLocationStyleProps, IDocumentCardLocationStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardLogo>`
  Gets the component ref.
- **logoIcon** `string`
  Describes DocumentCard Logo badge.

- **logoName** `string`
  Describe Logo name, optional.
- **styles** `IStyleFunctionOrObject<IDocumentCardLogoStyleProps, IDocumentCardLogoStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardPreview>`
  Gets the component ref.
- **getOverflowDocumentCountText** `(overflowCount: number) => string`
  The function return string that will describe the number of overflow documents. such as (overflowCount: number) =\> `+${ overflowCount } more`,
- **previewImages** `IDocumentCardPreviewImage[]`
  One or more preview images to display.
- **styles** `IStyleFunctionOrObject<IDocumentCardPreviewStyleProps, IDocumentCardPreviewStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **accentColor** `string`
  Hex color value of the line below the card, which should correspond to the document type. This should only be supplied when using the 'compact' card layout.

Deprecated at v4.17.1, to be removed at \>= v5.0.0.

- **children** `React.ReactNode`
  Child components to render within the card.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCard>`
  Optional callback to access the IDocumentCard interface. Use this instead of ref for accessing the public methods and properties of the component.
- **onClick** `(ev?: React.SyntheticEvent<HTMLElement>) => void`
  Function to call when the card is clicked or keyboard Enter/Space is pushed.
- **onClickHref** `string`
  A URL to navigate to when the card is clicked. If a function has also been provided, it will be used instead of the URL.
- **onClickTarget** `string`
  A target browser context for opening the link. If not specified, will open in the same tab/window.
- **role** `string`
  Aria role assigned to the documentCard (Eg. button, link). Use this to override the default assignment.
- **styles** `IStyleFunctionOrObject<IDocumentCardStyleProps, IDocumentCardStyles>`
  Call to provide customized styling that will layer on top of the variant rules

- **theme** `ITheme`
  Theme provided by HOC.
- **type** `DocumentCardType`
  The type of DocumentCard to display.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardStatus>`
  Gets the component ref.
- **status** `string`
  Describe status information. Required field.
- **statusIcon** `string`
  Describes DocumentCard status icon.
- **styles** `IStyleFunctionOrObject<IDocumentCardStatusStyleProps, IDocumentCardStatusStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **className** `string`
  Optional override class name
- **componentRef** `IRefObject<IDocumentCardTitle>`
  Gets the component ref.
- **shouldTruncate** `boolean`
  Whether we truncate the title to fit within the box. May have a performance impact.
- **showAsSecondaryTitle** `boolean`
  Whether show as title as secondary title style such as smaller font and lighter color.
- **styles** `IStyleFunctionOrObject<IDocumentCardTitleStyleProps, IDocumentCardTitleStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.
- **title** `string`
  Title text. If the card represents more than one document, this should be the title of one document and a "+X" string. For example, a collection of four documents would have a string of "Document.docx +3".

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use this control to represent Office documents or other files in aggregate views, such as when you're showing someone's most trending document.
- Incorporate metadata that is relevant and useful in this particular view. A card can be specialized to be about the document's latest changes, or about the document's popularity, as you see fit.

- Use a document card when you're illustrating an event that encompasses multiple files. For example, a card can be configured to represent a single upload action when adding more than one file.
- Don't use a document card in views where someone is likely to be performing bulk operations in files, or when the list may get very long. Specifically, if you're showing all the items inside an actual folder, a card may be overkill because the majority of the items in the folder may not have interesting metadata.
- Don't use a document card if space is at a premium or you can't show relevant and timely commands or metadata. Cards are useful because they can expose on-item interactions like "Share" buttons or view counts. If your app does not need this, show a simple grid or list of items instead, which are easier to scan.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

title <- "Long_file_name_with_underscores_used_to_separate_all_of_the_words"

previewImages <- list(
  list(
    previewImageSrc = "https://picsum.photos/318/196",
    width = 318,
    height = 196
  )
)

ui <- function(id) {
  ns <- NS(id)
  DocumentCard(
    DocumentCardPreview(previewImages = previewImages),
    DocumentCardTitle(
      title = title,
      shouldTruncate = TRUE
    ),
    DocumentCardActivity(
      activity = "Created a few minutes ago",
      people = list(list(name = "Annie Lindqvist"))
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
```

```
library(shiny)
library(shiny.fluent)

# Using icons in DocumentCardActions
ui <- function(id) {
  previewImages <- list(
    list(
      previewImageSrc = "https://picsum.photos/318/196",
      width = 318,
      height = 200
    )
  )
  fluidPage(
    DocumentCard(
      DocumentCardPreview(previewImages = previewImages),
      DocumentCardTitle(
        title = "Card",
        shouldTruncate = TRUE
      ),
      DocumentCardActivity(
        activity = "2022-03-23",
        people = list(list(name = "Annie Lindqvist"))
      ),
      DocumentCardActions(
        actions = list(
          list(
            iconProps = list(iconName = "Share"),
            onClick = JS("function() { alert('share icon clicked') }")
          ),
          list(
            iconProps = list(iconName = "Pin"),
            onClick = JS("function() { alert('pin icon clicked') }")
          ),
          list(
            iconProps = list(iconName = "Ringer"),
            onClick = JS("function() { alert('ringer icon clicked') }")
          )
        )
      )
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Dropdown` *Dropdown*

Description

A dropdown menu is a list in which the selected item is always visible while other items are visible on demand by clicking a dropdown button.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Dropdown(...)

Dropdown.shinyInput(inputId, ..., value = defaultValue)

updateDropdown.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **defaultSelectedKeys** `string[] | number[]`
  Keys that will be initially used to set selected items. This prop is only used when `multiSelect` is true (use `defaultSelectedKey` for single select). Mutually exclusive with `selectedKeys`.
- **dropdownWidth** `number`
  Custom width for dropdown. If value is 0, width of the input field is used.
- **isDisabled** `boolean`
  Deprecated at v0.52.0, use `disabled` instead.
- **keytipProps** `IKeytipProps`
  Optional keytip for this dropdown
- **multiSelectDelimiter** `string`
  When multiple items are selected, this will be used to separate values in the dropdown input.

- **notifyOnReselect** `boolean`
  If true, onChange will still be called when an already-selected item is clicked again in single select mode. (Normally it would not be called in this case.)
- **onChange** `(event: React.FormEvent<HTMLDivElement>, option?: IDropdownOption, index?: number) => v`
  Callback for when the selected option changes.
- **onChanged** `(option: IDropdownOption, index?: number) => void`
- **onRenderCaretDown** `IRenderFunction<IDropdownProps>`
  Custom renderer for chevron icon
- **onRenderLabel** `IRenderFunction<IDropdownProps>`
  Custom renderer for the label.
- **onRenderPlaceholder** `IRenderFunction<IDropdownProps>`
  Custom renderer for placeholder text
- **onRenderPlaceHolder** `IRenderFunction<IDropdownProps>`
  Custom renderer for placeholder text
- **onRenderTitle** `IRenderFunction<IDropdownOption[]>`
  Custom renderer for selected option displayed in input
- **options** `IDropdownOption[]`
  Options for the dropdown. If using `defaultSelectedKey` or `defaultSelectedKeys`, options must be pure for correct behavior.
- **placeHolder** `string`
  Input placeholder text. Displayed until an option is selected.
- **responsiveMode** `ResponsiveMode`
  By default, the dropdown will render the standard way for screen sizes `large` and above, or in a panel on `small` and `medium` screens. Manually set this prop to override this behavior.
- **selectedKeys** `string[] | number[] | null`
  Keys of the selected items, only used when `multiSelect` is true (use `selectedKey` for single select). If you provide this, you must maintain selection state by observing onChange events and passing a new prop value in when changed. Passing null will clear the selection. Mutually exclusive with `defaultSelectedKeys`.
- **styles** `IStyleFunctionOrObject<IDropdownStyleProps, IDropdownStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by higher order component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a dropdown list when there are multiple choices that can be collapsed under one title, if the list of items is too long, or when space is constrained.

- Use a dropdown list when the selected option is more important than the alternatives (in contrast to radio buttons where all the choices are visible, putting equal emphasis on all options).

Content:

- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.
- The dropdown list label should describe what can be found in the menu.
- Use shortened statements or single words as list options.
- If there isn't a default option, use "Select an option" as placeholder text.
- If "None" is an option, include it.
- Write the choices using parallel construction. For example, start with the same part of speech or verb tense.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

options <- list(
  list(key = "A", text = "Option A"),
  list(key = "B", text = "Option B"),
  list(key = "C", text = "Option C")
)

ui <- function(id) {
  ns <- NS(id)
  div(
    Dropdown.shinyInput(ns("dropdown"), value = "A", options = options),
    textOutput(ns("dropdownValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$dropdownValue <- renderText({
      sprintf("Value: %s", input$dropdown)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Rendering headers and dividers inside dropdown
```

```
DropdownMenuItemType <- function(type) {
  JS(paste0("jsmodule['@fluentui/react'].DropdownMenuItemType."), type)
}

ui <- function(id) {
  fluentPage(
    Dropdown(
      "fruit",
      label = "Fruit",
      multiSelect = TRUE,
      options = list(
        list(
          key = "fruitsHeader",
          text = "Fruit",
          itemType = DropdownMenuItemType("Header")
        ),
        list(key = "apple", text = "Apple"),
        list(key = "banana", text = "Banana"),
        list(key = "orange", text = "Orange", disabled = TRUE),
        list(key = "grape", text = "Grape"),
        list(
          key = "divider_1",
          text = "-",
          itemType = DropdownMenuItemType("Divider")
        ),
        list(
          key = "vegetablesHeader",
          text = "Vegetables",
          itemType = DropdownMenuItemType("Header"
          )
        ),
        list(key = "broccoli", text = "Broccoli"),
        list(key = "carrot", text = "Carrot"),
        list(key = "lettuce", text = "Lettuce")
      )
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

Facepile *Facepile*

Description

A face pile (`Facepile`) displays a list of personas. Each circle represents a person and contains their image or initials. Often this control is used when sharing who has access to a specific view or file, or when assigning someone a task within a workflow.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Facepile(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **addButtonProps** `IButtonProps`
  Button properties for the add face button
- **ariaDescription** `string`
  ARIA label for persona list
- **ariaLabel** `string`
  Defines the aria label that the screen readers use when focus goes on a list of personas.
- **chevronButtonProps** `IButtonProps`
  Deprecated at v0.70, use `overflowButtonProps` instead.
- **className** `string`
  Additional css class to apply to the Facepile
- **componentRef** `IRefObject<IFacepile>`
  Optional callback to access the IFacepile interface. Use this instead of ref for accessing the public methods and properties of the component.
- **getPersonaProps** `(persona: IFacepilePersona) => IPersonaSharedProps`
  Method to access properties on the underlying Persona control
- **maxDisplayablePersonas** `number`
  Maximum number of personas to show
- **onRenderPersona** `IRenderFunction<IFacepilePersona>`
  Optional custom renderer for the persona, gets called when there is one persona in personas array
- **onRenderPersonaCoin** `IRenderFunction<IFacepilePersona>`
  Optional custom renderer for the persona coins, gets called when there are multiple persona in personas array
- **overflowButtonProps** `IButtonProps`
  Properties for the overflow icon

- **overflowButtonType** `OverflowButtonType`
  Type of overflow icon to use
- **overflowPersonas** `IFacepilePersona[]`
  Personas to place in the overflow
- **personas** `IFacepilePersona[]`
  Array of IPersonaProps that define each Persona.
- **personaSize** `PersonaSize`
  Size to display the personas
- **showAddButton** `boolean`
  Show add person button
- **styles** `IStyleFunctionOrObject<IFacepileStyleProps, IFacepileStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- The face pile empty state should only include an "Add" button. Another variant is to use an input field with placeholder text instructing people to add a person. See the people picker component for the menu used to add people to the face pile list.
- When there is only one person in the face pile, consider using their name next to the face or initials.
- When there is a need to show the face pile expanded into a vertical list, include a downward chevron button. Selecting the chevron opens a standard list view of people.
- When the face pile exceeds a max number of 5 people, show a button at the end of the list indicating how many are not being shown. Clicking or tapping on the overflow would open a standard list view of people.
- The component can include an "Add" button which can be used for quickly adding a person to the list.
- When hovering over a person in the face pile, include a tooltip or people card that offers more information about that person.

Examples

```
library(shiny)
library(shiny.fluent)

personas <- list(
  list(personaName = "Adams Baker"),
  list(personaName = "Clark Davis"),
  list(personaName = "Evans Frank")
)
```

```
ui <- function(id) {
  ns <- NS(id)
  Facepile(personas = personas)
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`fluentPage` *Basic Fluent UI page*

Description

Creates a Fluent UI page with sensible defaults (included Fabric CSS classes, proper class given to the body tag, suppressed Bootstrap).

Usage

```
fluentPage(..., suppressBootstrap = TRUE)
```

Arguments

`...`	The contents of the document body.
`suppressBootstrap`	Whether to suppress Bootstrap.

Details

The Bootstrap library is suppressed by default, as it doesn't work well with Fluent UI in general.

Value

Object which can be passed as the UI of a Shiny app.

fluentPeople *A dataset of sample people based on Fluent UI examples*

Description

A dataset of sample people based on Fluent UI examples

Usage

```
fluentPeople
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 7 rows and 11 columns.

Source

https://developer.microsoft.com/en-us/fluentui#/controls/web/peoplepicker

fluentSalesDeals *A randomly generated dataset of imaginary sales deals*

Description

Sales deals to Top 10 companies from the Fortune 500 dataset (located at https://hifld-geoplatform.opendata.arcgis.com/datasets/fortune-500-corporate-headquarters) are randomly generated for each person in `fluentPeople`.

Usage

```
fluentSalesDeals
```

Format

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 100 rows and 30 columns.

`FocusTrapCallout` *FocusTrapZone*

Description

FocusTrapZone is used to trap the focus in any html element. Pressing tab will circle focus within the inner focusable elements of the FocusTrapZone.

Note: Trapping focus will restrict interaction with other elements in the website such as the side nav. Turn off the "Use trap zone" toggle control to allow this interaction to happen again.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
FocusTrapCallout(...)

FocusTrapZone(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaLabelledBy** `string`
  Sets the aria-labelledby attribute.
- **componentRef** `IRefObject<IFocusTrapZone>`
  Optional callback to access the IFocusTrapZone interface. Use this instead of ref for accessing the public methods and properties of the component.
- **disabled** `boolean`
  Whether to disable the FocusTrapZone's focus trapping behavior.
- **disableFirstFocus** `boolean`
  Do not put focus onto the first element inside the focus trap zone.
- **elementToFocusOnDismiss** `HTMLElement`
  Sets the element to focus on when exiting the FocusTrapZone.
- **enableAriaHiddenSiblings** `boolean`
  Puts aria-hidden=true on all non-ancestors of the current element, for screen readers. This is an experimental feature that will be graduated to default behavior after testing. This flag will be removed with the next major release.
- **firstFocusableSelector** `string | (() => string)`
  Class name (not actual selector) for first focusable item. Do not append a dot. Only applies if `focusPreviouslyFocusedInnerElement` is false.

- **focusPreviouslyFocusedInnerElement** `boolean`
  Specifies which descendant element to focus when `focus()` is called. If false, use the first focusable descendant, filtered by the `firstFocusableSelector` property if present. If true, use the element that was focused when the trap zone last had a focused descendant (or fall back to the first focusable descendant if the trap zone has never been focused).
- **forceFocusInsideTrap** `boolean`
  Whether the focus trap zone should force focus to stay inside of it.
- **ignoreExternalFocusing** `boolean`
  If false (the default), the trap zone will restore focus to the element which activated it once the trap zone is unmounted or disabled. Set to true to disable this behavior.
- **isClickableOutsideFocusTrap** `boolean`
  Whether clicks are allowed outside this FocusTrapZone.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  reactOutput(ns("focusTrapZone"))
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    output$focusTrapZone <- renderReact({
      useTrapZone <- isTRUE(input$useTrapZone)
      stackStyles <- list(root = list(
        border = if (useTrapZone) '2px solid #ababab' else 'transparent',
        padding = 10
      ))
      textFieldStyles <- list(root = list(width = 300));
      stackTokens = list(childrenGap = 8);

      div(
        FocusTrapZone(
          disabled = !useTrapZone,
          Stack(
            horizontalAlign = "start",
            tokens = stackTokens,
            styles = stackStyles,
            Toggle.shinyInput(ns("useTrapZone"),
              value = FALSE,
              label = "Use trap zone",
              onText = "On (toggle to exit)",
              offText = "Off (toggle to trap focus)"
```

```
            ),
            TextField.shinyInput(
              ns("textInput"),
              label = "Input inside trap zone",
              styles = textFieldStyles
            ),
            Link(
              href = "https://bing.com",
              target = "_blank",
              "Hyperlink inside trap zone"
            )
          )
        ),
        Link(
          href = "https://bing.com",
          target = "_blank",
          "Hyperlink outside trap zone"
        )
      )
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`FocusZone` *FocusZone*

Description

FocusZones abstract arrow key navigation behaviors. Tabbable elements (buttons, anchors, and elements with data-is-focusable='true' attributes) are considered when pressing directional arrow keys and focus is moved appropriately. Tabbing to a zone sets focus only to the current "active" element, making it simple to use the tab key to transition from one zone to the next, rather than through every focusable element.

Using a FocusZone is simple. Just wrap a bunch of content inside of a FocusZone, and arrows and tabbling will be handled for you! See examples below.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
FocusZone(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **allowFocusRoot** `boolean`
  Allows focus to park on root when focus is in the `FocusZone` at render time.
- **allowTabKey** `boolean`
  Allows tab key to be handled to tab through a list of items in the focus zone, an unfortunate side effect is that users will not be able to tab out of the focus zone and have to hit escape or some other key.
- **ariaDescribedBy** `string`
  Sets the aria-describedby attribute.
- **ariaLabelledBy** `string`
  Sets the aria-labelledby attribute.
- **as** `React.ElementType`
  A component that should be used as the root element of the FocusZone component.
- **checkForNoWrap** `boolean`
  Determines whether to check for data-no-horizontal-wrap or data-no-vertical-wrap attributes when determining how to move focus
- **className** `string`
  Additional class name to provide on the root element, in addition to the ms-FocusZone class.
- **componentRef** `IRefObject<IFocusZone>`
  Optional callback to access the IFocusZone interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultActiveElement** `string`
  Optionally provide a selector for identifying the initial active element.
- **defaultTabbableElement** `string | ((root: HTMLElement) => HTMLElement)`
  Optionally defines the initial tabbable element inside the FocusZone. If a string is passed then it is treated as a selector for identifying the initial tabbable element. If a function is passed then it uses the root element as a parameter to return the initial tabbable element.
- **direction** `FocusZoneDirection`
  Defines which arrows to react to.
- **disabled** `boolean`
  If set, the FocusZone will not be tabbable and keyboard navigation will be disabled. This does not affect disabled attribute of any child.
- **doNotAllowFocusEventToPropagate** `boolean`
  Whether the FocusZone should allow focus events to propagate past the FocusZone.
- **elementType** `any`
  Element type the root element will use. Default is "div".
- **handleTabKey** `FocusZoneTabbableElements`
  Allows tab key to be handled to tab through a list of items in the focus zone, an unfortunate side effect is that users will not be able to tab out of the focus zone and have to hit escape or some other key.

- **isCircularNavigation** `boolean`
  If set, will cycle to the beginning of the targets once the user navigates to the next target while at the end, and to the end when navigate to the previous at the beginning.
- **isInnerZoneKeystroke** `(ev: React.KeyboardEvent<HTMLElement>) => boolean`
  If provided, this callback will be executed on keypresses to determine if the user intends to navigate into the inner zone. Returning true will ask the first inner zone to set focus.
- **onActiveElementChanged** `(element?: HTMLElement, ev?: React.FocusEvent<HTMLElement>) => void`
  Callback for when one of immediate children elements gets active by getting focused or by having one of its respective children elements focused.
- **onBeforeFocus** `(childElement?: HTMLElement) => boolean`
  Callback method for determining if focus should indeed be set on the given element.
- **onFocus** `(event: React.FocusEvent<HTMLElement | FocusZone>) => void`
  Callback called when "focus" event triggered in FocusZone.
- **onFocusNotification** `() => void`
  Callback to notify creators that focus has been set on the FocusZone
- **pagingSupportDisabled** `boolean`
  Determines whether to disable the paging support for Page Up and Page Down keyboard scenarios.
- **preventDefaultWhenHandled** `boolean`
  If true, FocusZone prevents the default behavior of Keyboard events when changing focus between elements.
- **preventFocusRestoration** `boolean`
  If true, prevents the FocusZone from attempting to restore the focus to the inner element when the focus is on the root element after componentDidUpdate.
- **rootProps** `React.HTMLAttributes<HTMLDivElement>`
  Deprecated at v1.12.1. DIV props provided to the FocusZone will be mixed into the root element.
- **shouldEnterInnerZone** `(ev: React.KeyboardEvent<HTMLElement>) => boolean`
  Callback function that will be executed on keypresses to determine if the user intends to navigate into the inner (nested) zone. Returning true will ask the first inner zone to set focus.
- **shouldFocusInnerElementWhenReceivedFocus** `boolean`
  If true and FocusZone's root element (container) receives focus, the focus will land either on the defaultTabbableElement (if set) or on the first tabbable element of this FocusZone. Usually a case for nested focus zones, when the nested focus zone's container is a focusable element.
- **shouldFocusOnMount** `boolean`
  Determines if a default tabbable element should be force focused on FocusZone mount. @default false
- **shouldInputLoseFocusOnArrowKey** `(inputElement: HTMLInputElement) => boolean`
  A callback method to determine if the input element should lose focus on arrow keys
- **shouldRaiseClicks** `boolean`
  Determines whether the FocusZone will walk up the DOM trying to invoke click callbacks on focusable elements on Enter and Space keydowns to ensure accessibility for tags that don't guarantee this behavior.
- **shouldReceiveFocus** `(childElement?: HTMLElement) => boolean`
  Callback method for determining if focus should indeed be set on the given element.

- **shouldResetActiveElementWhenTabFromZone** `boolean`
  If true and TAB key is not handled by FocusZone, resets current active element to null value. For example, when roving index is not desirable and focus should always reset to the default tabbable element.
- **stopFocusPropagation** `boolean`
  Whether the FocusZone should allow focus events to propagate past the FocusZone.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

tokens <- list(childrenGap = 20)

ui <- function(id) {
  ns <- NS(id)
  Stack(
    tokens = tokens,
    horizontalAlign = "start",
    FocusZone(
      Stack(
        tokens = tokens,
        horizontal = TRUE,
        verticalAlign = "center",
        tags$span("Enabled FocusZone:"),
        DefaultButton(text = "Button 1"),
        DefaultButton(text = "Button 2"),
        TextField(placeholder = "FocusZone TextField"),
        DefaultButton(text = "Button 3")
      )
    ),
    DefaultButton(text = "Tabbable Element 1"),
    FocusZone(
      disabled = TRUE,
      Stack(
        tokens = tokens,
        horizontal = TRUE,
        verticalAlign = "center",
        tags$span("Disabled FocusZone:"),
        DefaultButton(text = "Button 1"),
        DefaultButton(text = "Button 2")
      )
    ),
    TextField(placeholder = "Tabbable Element 2")
  )
}

server <- function(id) {
```

```
  moduleServer(id, function(input, output, session) {})
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`FontIcon` *Icon*

Description

In a user interface, an icon is an image that represents an application, a capability, or some other concept or specific entity with meaning for the user. An icon is usually selectable but can also be a nonselectable image, such as a company's logo.

For a list of icons, visit our icon documentation.

Note that icons are not bundled by default and typically must be loaded by calling `initializeIcons` from the `@uifabric/icons` package at the root of your application. See the icon documentation for more details.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
FontIcon(...)

Icon(...)

ImageIcon(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Custom class to style the icon.
- **iconName** `string`
  The name of the icon to use from the icon font. If string is empty, a placeholder icon will be rendered the same width as an icon.
- **ariaLabel** `string`
  The aria label of the icon for the benefit of screen readers.

- **iconName** `string`
  The name of the icon to use from the icon font. If string is empty, a placeholder icon will be rendered the same width as an icon.
- **iconType** `IconType`
  The type of icon to render (image or icon font).
- **imageErrorAs** `React.ComponentType<IImageProps>`
  If rendering an image icon, this component will be rendered in the event that loading the image fails.
- **imageProps** `IImageProps`
  If rendering an image icon, these props will be passed to the Image component.
- **styles** `IStyleFunctionOrObject<IIconStyleProps, IIconStyles>`
  Gets the styles for an Icon.
- **theme** `ITheme`

- **className** `string`
  Custom class to style the icon.
- **imageProps** `IImageProps`
  Props passed to the Image component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

style <- list(fontSize = 50, margin = 10)

ui <- function(id) {
  ns <- NS(id)
  tags$div(
    FontIcon(iconName = "CompassNW", style = style),
    FontIcon(iconName = "Dictionary", style = style),
    FontIcon(iconName = "TrainSolid", style = style)
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`GroupedList` *GroupedList*

Description

A grouped list (`GroupedList`) allows you to render a set of items as multiple lists with various grouping properties.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
GroupedList(...)

GroupHeader(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Custom className
- **compact** `boolean`
  Boolean value to indicate if the component should render in compact mode. Set to false by default
- **componentRef** `IRefObject<{}>`
- **expandButtonProps** `React.HTMLAttributes<HTMLButtonElement>`
  Props for expand/collapse button
- **footerText** `string`
  Text to display for the group footer.
- **group** `IGroup`
  The group to be rendered by the header.
- **groupIndex** `number`
  The index of the group.
- **groupLevel** `number`
  The indent level of the group.
- **groups** `IGroup[]`
  Stores parent group's children.

- **indentWidth** `number`
  Width corresponding to a single level. This is multiplied by the groupLevel to get the full spacer width for the group.
- **isCollapsedGroupSelectVisible** `boolean`
  Determines if the group selection check box is shown for collapsed groups.
- **isGroupLoading** `(group: IGroup) => boolean`
  Callback to determine if a group has missing items and needs to load them from the server.
- **isSelected** `boolean`
  Deprecated at v.65.1 and will be removed by v 1.0. Use `selected` instead.
- **loadingText** `string`
  Text shown on group headers to indicate the group is being loaded.
- **onGroupHeaderClick** `(group: IGroup) => void`
  Callback for when the group header is clicked.
- **onRenderTitle** `IRenderFunction<IGroupHeaderProps>`
  Override which allows the caller to provider a custom renderer for the GroupHeader title.
- **onToggleCollapse** `(group: IGroup) => void`
  Callback for when the group is expanded or collapsed.
- **onToggleSelectGroup** `(group: IGroup) => void`
  Callback for when the group is selected.
- **onToggleSummarize** `(group: IGroup) => void`
  Callback for when the group "Show All" link is clicked
- **selected** `boolean`
  If all items in the group are selected.
- **selectionMode** `SelectionMode`
  The selection mode of the list the group lives within.
- **showAllLinkText** `string`
  Text to display for the group "Show All" link.
- **theme** `ITheme`
  Theme provided by the Higher Order Component
- **viewport** `IViewport`
  A reference to the viewport in which the header is rendered.
- **className** `string`
  Optional class name to add to the root element.
- **compact** `boolean`
  Boolean value to indicate if the component should render in compact mode. Set to false by default
- **componentRef** `IRefObject<IGroupedList>`
  Optional callback to access the IGroupedList interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dragDropEvents** `IDragDropEvents`
  Map of callback functions related to drag and drop functionality.
- **dragDropHelper** `IDragDropHelper`
  helper to manage drag/drop across item and groups

- **eventsToRegister** `{ eventName: string; callback: (context: IDragDropContext, event?: any) => void; }`
  Event names and corresponding callbacks that will be registered to groups and rendered elements
- **focusZoneProps** `IFocusZoneProps`
  Optional properties to pass through to the FocusZone.
- **getGroupHeight** `(group: IGroup, groupIndex: number) => number`
  Optional function to override default group height calculation used by list virtualization.
- **groupProps** `IGroupRenderProps`
  Optional override properties to render groups.
- **groups** `IGroup[]`
  Optional grouping instructions.
- **items** `any[]`
  List of items to render.
- **listProps** `IListProps`
  Optional properties to pass through to the list components being rendered.
- **onGroupExpandStateChanged** `(isSomeGroupExpanded: boolean) => void`
  Optional callback when the group expand state changes between all collapsed and at least one group is expanded.
- **onRenderCell** `(nestingDepth?: number, item?: any, index?: number) => React.ReactNode`
  Rendering callback to render the group items.
- **onShouldVirtualize** `(props: IListProps) => boolean`
  Optional callback to determine whether the list should be rendered in full, or virtualized. Virtualization will add and remove pages of items as the user scrolls them into the visible range. This benefits larger list scenarios by reducing the DOM on the screen, but can negatively affect performance for smaller lists. The default implementation will virtualize when this callback is not provided.
- **selection** `ISelection`
  Optional selection model to track selection state.
- **selectionMode** `SelectionMode`
  Controls how/if the list manages selection.
- **styles** `IStyleFunctionOrObject<IGroupedListStyleProps, IGroupedListStyles>`
  Style function to be passed in to override the themed or default styles
- **theme** `ITheme`
  Theme that is passed in from Higher Order Component
- **usePageCache** `boolean`
  boolean to control if pages containing unchanged items should be cached, this is a perf optimization The same property in List.Props
- **viewport** `IViewport`
  Optional Viewport, provided by the parent component.
- **styles** `IStyleFunctionOrObject<IGroupFooterStyleProps, IGroupFooterStyles>`
  Style function to be passed in to override the themed or default styles
- **checked** `boolean`

- **theme** `ITheme`

- **ariaPosInSet** `number`
  Defines an element's number or position in the current set of listitems or treeitems
- **ariaSetSize** `number`
  Defines the number of items in the current set of listitems or treeitems
- **expandButtonIcon** `string`
  Defines the name of a custom icon to be used for group headers. If not set, the default icon will be used
- **expandButtonProps** `React.HTMLAttributes<HTMLButtonElement>`
  Native props for the GroupHeader expand and collapse button
- **groupedListId** `string`
  GroupedList id for aria-controls
- **onRenderGroupHeaderCheckbox** `IRenderFunction<IGroupHeaderCheckboxProps>`
  If provided, can be used to render a custom checkbox
- **selectAllButtonProps** `React.HTMLAttributes<HTMLButtonElement>`
  Native props for the GroupHeader select all button
- **styles** `IStyleFunctionOrObject<IGroupHeaderStyleProps, IGroupHeaderStyles>`
  Style function to be passed in to override the themed or default styles
- **useFastIcons** `boolean`
  Whether to use fast icon and check components. The icons can't be targeted by customization but are still customizable via class names.
- **collapseAllVisibility** `CollapseAllVisibility`
  Flag to indicate whether to ignore the collapsing icon on header.
- **footerProps** `IGroupFooterProps`
  Information to pass in to the group footer.
- **getGroupItemLimit** `(group: IGroup) => number`
  Grouping item limit.
- **headerProps** `IGroupHeaderProps`
  Information to pass in to the group header.
- **isAllGroupsCollapsed** `boolean`
  Boolean indicating if all groups are in collapsed state.
- **onRenderFooter** `IRenderFunction<IGroupFooterProps>`
  Override which allows the caller to provide a custom footer.
- **onRenderHeader** `IRenderFunction<IGroupHeaderProps>`
  Override which allows the caller to provide a custom header.
- **onRenderShowAll** `IRenderFunction<IGroupShowAllProps>`
  Override which allows the caller to provide a custom Show All link.
- **onToggleCollapseAll** `(isAllCollapsed: boolean) => void`
  Callback for when all groups are expanded or collapsed.
- **role** `string`
  Override which allows the caller to provide a custom aria role

- **showAllProps** `IGroupShowAllProps`
  Information to pass in to the group Show all footer.
- **showEmptyGroups** `boolean`
  Boolean indicating if empty groups are shown
- **showAllLinkText** `string`
  The Show All link text.
- **styles** `IStyleFunctionOrObject<IGroupShowAllStyleProps, IGroupShowAllStyles>`
  Style function to be passed in to override the themed or default styles
- **count** `number`
  Count of spacer(s)
- **indentWidth** `number`
  How much to indent
- **styles** `IStyleFunctionOrObject<IGroupSpacerStyleProps, IGroupSpacerStyles>`
  Style function to be passed in to override the themed or default styles
- **theme** `ITheme`
  Theme from Higher Order Component

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

FAQ:

My List is not re-rendering when I mutate its items. What should I do?:
To determine if the list within the grouped list should re-render its contents, the component performs a referential equality check within its `shouldComponentUpdate` method. This is done to minimize the performance overhead associating with re-rendering the virtualized List pages, as recommended by the React documentation.
As a result of this implementation, the inner list will not determine it should re-render if the array values are mutated. To avoid this problem, we recommend re-creating the items array backing the grouped list by using a method such as `Array.prototype.concat` or ES6 spread syntax shown below:

```
public appendItems(): void {
  const { items } = this.state;

  this.setState({
    items: [...items, ...['Foo', 'Bar']]
  })
}

public render(): JSX.Element {
  const { items } = this.state;

  return <GroupedList items={items} />;
}
```

By re-creating the items array without mutating the values, the inner List will correctly determine its contents have changed and then it should re-render with the new values.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  GroupedList(
    items = list("Item A", "Item B", "Item C", "Item D", "Item E"),
    groups = list(
      list(key = "g1", name = "Some items", startIndex = 0, count = 2),
      list(key = "g2", name = "More items", startIndex = 2, count = 3)
    ),
    selectionMode = 0,
    onRenderCell = JS("(depth, item) => (
      jsmodule['react'].createElement('span', { style: { paddingLeft: 49 } }, item)
    )")
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  fluentPage(
    GroupedList(
      items = list("Item A", "Item B", "Item C", "Item D", "Item E"),
      groups = list(
        list(key = "g1", name = "Some items", startIndex = 0, count = 2),
        list(key = "g2", name = "More items", startIndex = 2, count = 3)
      ),
      selectionMode = 0,
      onRenderCell = JS(
        "(depth, item) => (
          jsmodule['react'].createElement('span', { style: { paddingLeft: 50 } }, item)
        )"
      ),
      groupProps = list(
        onRenderHeader = JS(
          "(props) => (
```

```
            jsmodule['react'].createElement(
              jsmodule['@fluentui/react'].GroupHeader,
              { ...props, styles: { headerCount: { display: 'none' } } },
              props
            )
          )"
        )
      )
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`HoverCard` *HoverCard*

Description

Hover cards (`HoverCard`) show commands and information, such as metadata and activity, when someone hovers over an item.

- Tabbing with a keyboard to the element triggering the HoverCard to open on focus (see first example). In this case no further navigation within the card is available and navigating to the next element will close the card.
- Tabbing with a keyboard to the element triggering the HoverCard and opening when the hotKey is pressed (see second example). When the card is opened it will automatically focus the first focusable element of the card content.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
HoverCard(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **compactCardHeight** `number`
  Height of compact card
- **expandedCardHeight** `number`
  Height of expanded card
- **mode** `ExpandingCardMode`
  Use to open the card in expanded format and not wait for the delay
- **onRenderCompactCard** `IRenderFunction<any>`
  Render function to populate compact content area
- **onRenderExpandedCard** `IRenderFunction<any>`
  Render function to populate expanded content area
- **cardDismissDelay** `number`
  Length of card dismiss delay. A min number is necessary for pointer to hop between target and card
- **cardOpenDelay** `number`
  Length of compact card delay
- **className** `string`
  Additional CSS class(es) to apply to the HoverCard root element.
- **componentRef** `IRefObject<IHoverCard>`
  Optional callback to access the IHoverCardHost interface. Use this instead of ref for accessing the public methods and properties of the component.
- **eventListenerTarget** `HTMLElement | string | null`
  This prop is to separate the target to anchor hover card from the target to attach event listener. If set, this prop separates the target to anchor the hover card from the target to attach the event listener. When no `eventListenerTarget` given, HoverCard will use `target` prop or its root to set event listener.
- **expandedCardOpenDelay** `number`
  Time in ms when expanded card should open after compact card
- **expandingCardProps** `IExpandingCardProps`
  Additional ExpandingCard props to pass through HoverCard like renderers, target. gapSpace etc. Used along with 'type' prop set to HoverCardType.expanding. Reference detail properties in ICardProps and IExpandingCardProps.
- **instantOpenOnClick** `boolean`
  Enables instant open of the full card upon click
- **onCardExpand** `() => void`
  Callback when visible card is expanded.
- **onCardHide** `() => void`
  Callback when card hides
- **onCardVisible** `() => void`
  Callback when card becomes visible
- **openHotKey** `KeyCodes`
  HotKey used for opening the HoverCard when tabbed to target.

- **plainCardProps** `IPlainCardProps`
  Additional PlainCard props to pass through HoverCard like renderers, target, gapSpace etc. Used along with 'type' prop set to HoverCardType.plain. See for more details ICardProps and IPlainCardProps interfaces.
- **setAriaDescribedBy** `boolean`
  Whether or not to mark the container as described by the hover card. If not specified, the caller should mark as element as described by the hover card id.
- **setInitialFocus** `boolean`
  Set to true to set focus on the first focusable element in the card. Works in pair with the 'trapFocus' prop.
- **shouldBlockHoverCard** `() => void`
  Should block hover card or not
- **sticky** `boolean`
  If true disables Card dismiss upon mouse leave, so that card sticks around.
- **styles** `IStyleFunctionOrObject<IHoverCardStyleProps, IHoverCardStyles>`
  Custom styles for this component
- **target** `HTMLElement | string | null`
  Optional target element to tag hover card on. If not provided and using HoverCard as a wrapper, don't set the 'data-is-focusable=true' attribute to the root of the wrapped child. If no target is given, HoverCard will use its root as a target and become the focusable element with a focus listener attached to it.
- **theme** `ITheme`
  Theme provided by higher order component.
- **trapFocus** `boolean`
  Set to true if you want to render the content of the HoverCard in a FocusTrapZone for accessibility reasons. Optionally 'setInitialFocus' prop can be set to true to move focus inside the FocusTrapZone.
- **type** `HoverCardType`
  Type of the hover card to render.
- **onRenderPlainCard** `IRenderFunction<any>`
  Render function to populate compact content area

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Hover cards contain both compact and expanded states, with the compact state appearing after 500 milliseconds and the expanded state appearing as the user continues to hover after 1500 milliseconds.
- The hover card positions itself automatically, depending upon where the target is on the viewport. Position the target so the card doesn't obstruct inline commanding on the item.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  HoverCard(
    type = "PlainCard",
    plainCardProps = JS("{
      onRenderPlainCard: (a, b, c) => 'HoverCard contents',
      style: { margin: 10 }
    }"),
    "Hover over me"
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Image` *Image*

Description

An image is a graphic representation of something (e.g photo or illustration). The borders have been added to these examples in order to help visualize empty space in the image frame.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Image(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the Component

- **coverStyle** `ImageCoverStyle`
  Specifies the cover style to be used for this image. If not specified, this will be dynamically calculated based on the aspect ratio for the image.
- **errorSrc** `string`
  Deprecated at v1.3.6, to replace the src in case of errors, use `onLoadingStateChange` instead and rerender the Image with a difference src.
- **imageFit** `ImageFit`
  Used to determine how the image is scaled and cropped to fit the frame.
- **maximizeFrame** `boolean`
  If true, the image frame will expand to fill its parent container.
- **onLoadingStateChange** `(loadState: ImageLoadState) => void`
  Optional callback method for when the image load state has changed. The 'loadState' parameter indicates the current state of the Image.
- **shouldFadeIn** `boolean`
  If true, fades the image in when loaded.
- **shouldStartVisible** `boolean`
  If true, the image starts as visible and is hidden on error. Otherwise, the image is hidden until it is successfully loaded. This disables shouldFadeIn.
- **styles** `IStyleFunctionOrObject<IImageStyleProps, IImageStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Image(src = "https://via.placeholder.com/350x150")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Keytip` *Keytip*

Description

A Keytip is a small popup near a component that indicates a key sequence that will trigger that component. These are not to be confused with keyboard shortcuts; they are instead key sequences to traverse through levels of UI components. Technically, a Keytip is a wrapper around a Callout where the target element is discovered through a 'data-ktp-target' attribute on that element.

To enable Keytips on your page, a developer will add the KeytipLayer component somewhere in their document. It can be added anywhere in your document, but must only be added once. Use the registerKeytip utility helper to add a Keytip. A user will enter and exit keytip mode with a IKeytipTransitionSequence, which is a key with any amount of modifiers (Alt, Shift, etc).

By default, the entry and exit sequence is 'Alt-Windows' (Meta) on Windows and 'Option-Control' on macOS. There is also a sequence to 'return' up a level of keytips while traversing. This is by default 'Esc'.

Fluent UI React components that have keytips enabled have an optional 'keytipProps' prop which handles registering, unregistering, and rendering of the keytip. The keySequences of the Keytip should be the full sequence to get to that keytip. There is a 'buildKeytipConfigMap' helper which will build a map of ID -> IKeytipProps to assist in defining your keytips.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

`KeytipLayer` *Keytips*

Description

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
KeytipLayer(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **componentRef** `IRefObject<IKeytipLayer>`
  Optional callback to access the KeytipLayer component. Use this instead of ref for accessing the public methods and properties of the component.
- **content** `string`
  String to put inside the layer to be used for the aria-describedby for the component with the keytip Should be one of the starting sequences
- **keytipExitSequences** `IKeytipTransitionKey[]`
  List of key sequences that will exit keytips mode
- **keytipReturnSequences** `IKeytipTransitionKey[]`
  List of key sequences that execute the return functionality in keytips (going back to the previous level of keytips)
- **keytipStartSequences** `IKeytipTransitionKey[]`
  List of key sequences that will start keytips mode
- **onEnterKeytipMode** `() => void`
  Callback function triggered when keytip mode is entered
- **onExitKeytipMode** `(ev?: React.KeyboardEvent<HTMLElement> | React.MouseEvent<HTMLElement>) => void`
  Callback function triggered when keytip mode is exited. ev is the Mouse or Keyboard Event that triggered the exit, if any.
- **styles** `IStyleFunctionOrObject<IKeytipLayerStyleProps, IKeytipLayerStyles>`
  (Optional) Call to provide customized styling.
- **calloutProps** `ICalloutProps`
  ICalloutProps to pass to the callout element
- **content** `string`
  Content to put inside the keytip
- **disabled** `boolean`
  T/F if the corresponding control for this keytip is disabled
- **hasDynamicChildren** `boolean`
  Whether or not this keytip will have children keytips that are dynamically created (DOM is generated on keytip activation). Common cases are a Pivot or Modal.
- **hasMenu** `boolean`
  Whether or not this keytip belongs to a component that has a menu Keytip mode will stay on when a menu is opened, even if the items in that menu have no keytips
- **keySequences** `string[]`
  Array of KeySequences which is the full key sequence to trigger this keytip Should not include initial 'start' key sequence
- **offset** `Point`
  Offset x and y for the keytip, added from the top-left corner By default the keytip will be anchored to the bottom-center of the element
- **onExecute** `(executeTarget: HTMLElement | null, target: HTMLElement | null) => void`
  Function to call when this keytip is activated. 'executeTarget' is the DOM element marked with 'data-ktp-execute-target'. 'target' is the DOM element marked with 'data-ktp-target'.

- **onReturn** `(executeTarget: HTMLElement | null, target: HTMLElement | null) => void`
  Function to call when the keytip is the currentKeytip and a return sequence is pressed. 'executeTarget' is the DOM element marked with 'data-ktp-execute-target'. 'target' is the DOM element marked with 'data-ktp-target'.
- **overflowSetSequence** `string[]`
  Full KeySequence of the overflow set button, will be set automatically if this keytip is inside an overflow
- **styles** `IStyleFunctionOrObject<IKeytipStyleProps, IKeytipStyles>`
  Optional styles for the component.
- **theme** `ITheme`
  Theme for the component
- **visible** `boolean`
  T/F if the keytip is visible

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

makeScript <- function(js) {
  tagList(
    shiny.react::reactDependency(),
    htmltools::htmlDependency(
      name = "KeytipsExample",
      version = "0", # Not used.
      src = c(href = ""), # Not used.
      head = paste0("<script>", js, "</script>")
    )
  )
}

ui <- function(id) {
  ns <- NS(id)
  tagList(
    makeScript(paste0("setTimeout(() => {
      const btnExecute = (el) => {
        el.click();
      };

      const keytipConfig = {
        keytips: [
          // Button example
          {
            id: 'Button',
            content: '1A',
            optionalProps: {
```

```
        onExecute: btnExecute,
      },
    },
    {
      id: 'CompoundButton',
      content: '1B',
      optionalProps: {
        onExecute: btnExecute,
      },
    },
    {
      id: 'ButtonWithMenu',
      content: '2A',
      optionalProps: {
        onExecute: btnExecute,
      },
      children: [
        {
          id: 'ButtonMenuItem1',
          content: 'E',
          optionalProps: {
            onExecute: btnExecute,
          },
        },
        {
          id: 'ButtonMenuItem2',
          content: '8',
          optionalProps: {
            onExecute: btnExecute,
          },
        },
      ],
    }
  ],
};

keytipMap = jsmodule['@fluentui/react'].buildKeytipConfigMap(keytipConfig);

window.buttonProps = {
  items: [
    {
      key: 'buttonMenuItem1',
      text: 'Menu Item 1',
      keytipProps: keytipMap.ButtonMenuItem1,
      onClick: () => Shiny.setInputValue('", ns("button3"), "', Math.random())
    },
    {
      key: 'buttonMenuItem2',
      text: 'Menu Item 2',
      keytipProps: keytipMap.ButtonMenuItem2,
      onClick: () => Shiny.setInputValue('", ns("button3"), "', Math.random())
    },
  ],
```

```
        };
      })")),
      textOutput(ns("keytipsResult")),
      div(
        Label(
          paste0(
         "To open keytips, hit 'Alt-Windows' on Windows/Linux and 'Option-Control' on macOS.",
         "Keytips will appear. Type what you see, e.g. 1 and then A to 'click' the first button."
          )
        ),
        Label(
          paste0(
            "When multiple Keytips start with the same character,",
            "typing that character will filter the visible keytips."
          )
        ),
        KeytipLayer(),
        Stack(horizontal = TRUE, tokens = list(childrenGap = 20),
          DefaultButton.shinyInput(
            ns("button1"),
            keytipProps = JS("keytipMap.Button"),
            text = "Button"
          ),
          CompoundButton.shinyInput(
            ns("button2"),
            style = list(marginBottom = 28),
            keytipProps = JS("keytipMap.CompoundButton"),
            text = "Compound Button",
            secondaryText = 'With a Keytip'
          ),
          DefaultButton.shinyInput(
            ns("button3"),
            keytipProps = JS("keytipMap.ButtonWithMenu"),
            text = "Button with Menu",
            menuProps = JS("buttonProps")
          )
        )
      )
    )
  }

  server <- function(id) {
    moduleServer(id, function(input, output, session) {
      clicks <- reactiveVal(0)
      addClick <- function() clicks(clicks() + 1)
      output$keytipsResult <- renderText(paste("Buttons clicked: ", clicks()))
      observeEvent(input$button1, addClick())
      observeEvent(input$button2, addClick())
      observeEvent(input$button3, addClick())
    })
  }

  if (interactive()) {
```

```
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Label` *Label*

Description

Labels give a name or title to a control or group of controls, including text fields, check boxes, combo boxes, radio buttons, and drop-down menus.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Label(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **as** `IComponentAs<React.AllHTMLAttributes<HTMLElement>>`
  Render the root element as another type.
- **componentRef** `IRefObject<ILabel>`
  Optional callback to access the ILabel interface. Use this instead of ref for accessing the public methods and properties of the component.
- **disabled** `boolean`
  Renders the label as disabled.
- **required** `boolean`
  Whether the associated form field is required or not
- **styles** `IStyleFunctionOrObject<ILabelStyleProps, ILabelStyles>`
  Styles for the label.
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Labels should be close to the control they're paired with.

Content:

- Labels should describe the purpose of the control.
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.
- Be short and concise.
- Use nouns or short noun phrases.
- Don't use labels as instructional text. For example, "Click to get started".

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Label("Required label", required = TRUE)
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

Layer *Layer*

Description

A Layer is a technical component that does not have specific Design guidance.

Layers are used to render content outside of a DOM tree, at the end of the document. This allows content to escape traditional boundaries caused by "overflow: hidden" css rules and keeps it on the top without using z-index rules. This is useful for example in ContextualMenu and Tooltip scenarios, where the content should always overlay everything else.

There are some special considerations. Due to the nature of rendering content elsewhere asynchronously, React refs within content will not be resolvable synchronously at the time the Layer is mounted. Therefore, to use refs correctly, use functional refs `ref={ (el) => { this._root = el; }` rather than string refs `ref='root'`. Additionally measuring the physical Layer element will not include any of the children, since it won't render it. Events that propgate from within the content will not go through the Layer element as well.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Layer(...)

LayerHost(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the Layer
- **componentRef** `IRefObject<ILayer>`
  Optional callback to access the ILayer interface. Use this instead of ref for accessing the public methods and properties of the component.
- **eventBubblingEnabled** `boolean`
  When enabled, Layer allows events to bubble up from Layer content. Traditionally Layer has not had this behavior. This prop preserves backwards compatibility by default while allowing users to opt in to the new event bubbling functionality.
- **hostId** `string`
  The optional id property provided on a LayerHost that this Layer should render within. The LayerHost does not need to be immediately available but once has been rendered, and if missing, we'll avoid trying to render the Layer content until the host is available. If an id is not provided, we will render the Layer content in a fixed position element rendered at the end of the document.
- **insertFirst** `boolean`
  Whether the layer should be added as the first child of the host. If true, the layer will be inserted as the first child of the host By default, the layer will be appended at the end to the host
- **onLayerDidMount** `() => void`
  Callback for when the layer is mounted.
- **onLayerMounted** `() => void`
  Callback for when the layer is mounted.
- **onLayerWillUnmount** `() => void`
  Callback for when the layer is unmounted.
- **styles** `IStyleFunctionOrObject<ILayerStyleProps, ILayerStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    style = "margin-top: 60px; border: 1px solid navy; padding: 10px; background: #eee;",
   Checkbox.shinyInput(ns("useLayer"), value = FALSE, label = "Display a message in a layer"),
    reactOutput(ns("layer"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$layer <- renderReact({
      box <- div(
        style = "background-color: #60C7FF; margin: 10px; padding: 10px",
        "Hello!"
      )
      if (isTRUE(input$useLayer)) Layer(box)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    LayerHost(id = "host", style = list(border = "1px dashed", padding = 10)),
    "Layer children are rendered in the LayerHost",
    Layer(hostId = "host", "Content")
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Link` *Link*

Description

Links lead to another part of an app, other pages, or help articles. They can also be used to initiate commands.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Link(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **as** `string | React.ComponentClass | React.FunctionComponent`
  A component that should be used as the root element of the link returned from the Link component.
- **componentRef** `IRefObject<ILink>`
  Optional callback to access the ILink interface. Use this instead of ref for accessing the public methods and properties of the component.
- **disabled** `boolean`
  Whether the link is disabled
- **keytipProps** `IKeytipProps`
  Optional keytip for this Link
- **styles** `IStyleFunctionOrObject<ILinkStyleProps, ILinkStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Links visually indicate that they can be clicked, typically by being displayed using the visited or unvisited link system colors. Traditionally, links are underlined as well, but that approach is often reserved for body copy links within an article.

Content:

- People should be able to accurately predict the result of selecting a link based on its link text and optional tooltip.
- Use descriptive, actionable link text when possible. Avoid using URLs as link text.
- Don't use if the action is destructive or irreversible. Links aren't appropriate for commands with significant consequences.
- Keep discrete links far enough apart that people can differentiate between them and easily select each one.
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Link(href = "https://appsilon.com", "Appsilon")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

List *List*

Description

A list provides a base component for rendering large sets of items. It's agnostic of layout, the tile component used, and selection management.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
List(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **divProps** `React.HTMLAttributes<HTMLDivElement>`
  Props to apply to the list root element.
- **pages** `IPage<T>[]`
  The active pages to be rendered into the list. These will have been rendered using `onRenderPage`.
- **rootRef** `React.Ref<HTMLDivElement>`
  The ref to be applied to the list root. The `List` uses this element to track scroll position and sizing.
- **surfaceElement** `JSX.Element | null`
  The content to be rendered as the list surface element. This will have been rendered using `onRenderSurface`.
- **divProps** `React.HTMLAttributes<HTMLDivElement>`
  Props to apply to the list surface element.
- **pageElements** `JSX.Element[]`
  The content to be rendered representing all active pages.
- **pages** `IPage<T>[]`
  The active pages to be rendered into the list. These will have been rendered using `onRenderPage`.
- **surfaceRef** `React.Ref<HTMLDivElement>`
  A ref to be applied to the surface element. The `List` uses this element to track content size and focus.
- **className** `string`
  Optional classname to append to root list.
- **componentRef** `IRefObject<IList>`
  Optional callback to access the IList interface. Use this instead of ref for accessing the public methods and properties of the component.
- **getItemCountForPage** `(itemIndex?: number, visibleRect?: IRectangle) => number`
  Method called by the list to get how many items to render per page from specified index. In general, use `getPageSpecification` instead.
- **getKey** `(item: T, index?: number) => string`
  Optional callback to get the item key, to be used on render.
- **getPageHeight** `(itemIndex?: number, visibleRect?: IRectangle, itemCount?: number) => number`
  Method called by the list to get the pixel height for a given page. By default, we measure the first page's height and default all other pages to that height when calculating the surface space. It is ideal to be able to adequately predict page heights in order to keep the surface space from jumping in pixels, which has been seen to cause browser performance issues. In general, use `getPageSpecification` instead.

- **getPageSpecification** `(itemIndex?: number, visibleRect?: IRectangle) => IPageSpecification`
  Called by the list to get the specification for a page. Use this method to provide an allocation of items per page, as well as an estimated rendered height for the page. The list will use this to optimize virtualization.
- **getPageStyle** `(page: IPage<T>) => any`
  Method called by the list to derive the page style object. For spacer pages, the list will derive the height and passed in heights will be ignored.
- **ignoreScrollingState** `boolean`
  Whether to disable scroll state updates. This causes the isScrolling arg in onRenderCell to always be undefined. This is a performance optimization to let List skip a render cycle by not updating its scrolling state.
- **items** `T[]`
  Items to render.
- **onPageAdded** `(page: IPage<T>) => void`
  Optional callback for monitoring when a page is added.
- **onPageRemoved** `(page: IPage<T>) => void`
  Optional callback for monitoring when a page is removed.
- **onPagesUpdated** `(pages: IPage<T>[]) => void`
  Optional callback invoked when List rendering completed. This can be on initial mount or on re-render due to scrolling. This method will be called as a result of changes in List pages (added or removed), and after ALL the changes complete. To track individual page Add / Remove use onPageAdded / onPageRemoved instead.
- **onRenderCell** `(item?: T, index?: number, isScrolling?: boolean) => React.ReactNode`
  Method to call when trying to render an item.
- **onRenderPage** `IRenderFunction<IPageProps<T>>`
  Called when the List will render a page. Override this to control how cells are rendered within a page.
- **onRenderRoot** `IRenderFunction<IListOnRenderRootProps<T>>`
  Render override for the element at the root of the `List`. Use this to apply some final attributes or structure to the content each time the list is updated with new active pages or items.
- **onRenderSurface** `IRenderFunction<IListOnRenderSurfaceProps<T>>`
  Render override for the element representing the surface of the `List`. Use this to alter the structure of the rendered content if necessary on each update.
- **onShouldVirtualize** `(props: IListProps<T>) => boolean`
  Optional callback to determine whether the list should be rendered in full, or virtualized. Virtualization will add and remove pages of items as the user scrolls them into the visible range. This benefits larger list scenarios by reducing the DOM on the screen, but can negatively affect performance for smaller lists. The default implementation will virtualize when this callback is not provided.
- **renderCount** `number`
  Number of items to render. Defaults to items.length.
- **renderedWindowsAhead** `number`
  In addition to the visible window, how many windowHeights should we render ahead.
- **renderedWindowsBehind** `number`
  In addition to the visible window, how many windowHeights should we render behind.

- **role** `string`
  The role to assign to the list root element. Use this to override the default assignment of 'list' to the root and 'listitem' to the cells.
- **startIndex** `number`
  Index in items array to start rendering from. Defaults to 0.
- **usePageCache** `boolean`
  Boolean value to enable render page caching. This is an experimental performance optimization that is off by default.
- **version** `{}`
  An object which can be passed in as a fresh instance to 'force update' the list.
- **page** `IPage<T>`
  The allocation data for the page.
- **role** `string`
  The role being assigned to the rendered page element by the list.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- List items are composed of selection, icon, and name columns at minimum. You can include other columns, such as date modified, or any other metadata field associated with the collection.
- Avoid using file type icon overlays to denote status of a file as it can make the entire icon unclear.
- If there are multiple lines of text in a column, consider the variable row height variant.
- Give columns ample default width to display information.

Content:

- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

FAQ:

My scrollable content isn't updating on scroll, what should I do?:
Add the data-is-scrollable="true" attribute to your scrollable element containing the List.
By default, the List will use the `<body>` element as the scrollable element. If you contain List within a scrollable `<div>` using overflow: auto or scroll, List needs to listen for scroll events on that element instead. On initialization, List will traverse up the DOM looking for the first element with the data-is-scrollable attribute to know which element to listen to for knowing when to re-evaulate the visible window.

My list isn't re-rendering when I mutate its items, what should I do?:
To determine if List should re-render its contents, the component performs a referential equality check on the items array in its shouldComponentUpdate method. This is done to minimize the performance overhead associating with re-rendering the virtualized list pages, as recommended

by the React documentation. As a result of this implementation, List will not determine it should re-render if values within the array are mutated. To avoid this problem, we recommend re-creating the items array using a method such as Array.prototype.concat or ES6 spread syntax shown below:

```
public appendItems(): void {
  const { items } = this.state;

  this.setState({
    items: [...items, ...[{ name: 'Foo' }, { name: 'Bar' }]]
  })
}

public render(): JSX.Element {
  const { items } = this.state;

  return <List items={items} />;
}
```

Since the items array has been re-created, the list will conclude that its contents have changed and it should re-render the new values.

How do I limit rendering to improve performance?:
Performance is important, and DOM content is expensive. Therefore, limit what you render. The list component applies this principle by using UI virtualization. Unlike a simple for loop that renders all items in a set, a list only renders a subset of items, and as you scroll around, the subset of rendered content is shifted. This gives a much better experience for large sets, especially when the per-item components are complex/render-intensive/network-intensive.
A list breaks down the set of items passed in into pages. Only pages within a "materialized window" are actually rendered. As that window changes due to scroll events, pages that fall outside that window are removed, and their layout space is remembered and pushed into spacer elements. This gives the user the experience of browsing massive amounts of content but only using a small number of actual elements. This gives the browser much less layout to resolve, and gives React DOM diffing much less content to worry about.
Note: If onRenderCell is not provided in IListProps, the list will attempt to render the name property for each object in the items array.

Examples

```
library(shiny)
library(shiny.fluent)

items <- do.call(paste0, replicate(20, sample(LETTERS, 200, TRUE), FALSE))

ui <- function(id) {
  ns <- NS(id)
  div(
    style = "overflow: auto; max-height: 400px",
    List(
      items = items,
      onRenderCell = JS("(item, index) => `${index} ${item}`")
    )
```

```
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`MarqueeSelection` *MarqueeSelection*

Description

The MarqueeSelection component provides a service which allows the user to drag a rectangle to be drawn around items to select them. This works in conjunction with a selection object, which can be used to generically store selection state, separate from a component that consumes the state.

MarqueeSelection also works in conjunction with the AutoScroll utility to automatically scroll the container when we drag a rectangle within the vicinity of the edges.

When a selection rectangle is dragged, we look for elements with the **data-selection-index** attribute populated. We get these elements' boundingClientRects and compare them with the root's rect to determine selection state. We update the selection state appropriately.

In virtualization cases where items that were once selected are dematerialized, we will keep the item in its previous state until we know definitively if it's on/off. (In other words, this works with List.)

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
MarqueeSelection(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional CSS class(es) to apply to the MarqueeSelection.
- **componentRef** `IRefObject<IMarqueeSelection>`
  Optional callback to access the IMarqueeSelection interface. Use this instead of ref for accessing the public methods and properties of the component.

- **isDraggingConstrainedToRoot** `boolean`
  Optional flag to restrict the drag rect to the root element, instead of allowing the drag rect to start outside of the root element boundaries.
- **isEnabled** `boolean`
  Optional flag to control the enabled state of marquee selection. This allows you to render it and have events all ready to go, but conditionally disable it. That way transitioning between enabled/disabled generate no difference in the DOM.
- **onShouldStartSelection** `(ev: MouseEvent) => boolean`
  Optional callback that is called, when the mouse down event occurs, in order to determine if we should start a marquee selection. If true is returned, we will cancel the mousedown event to prevent upstream mousedown handlers from executing.
- **rootProps** `React.HTMLAttributes<HTMLDivElement>`
  Optional props to mix into the root DIV element.
- **selection** `ISelection`
  The selection object to interact with when updating selection changes.
- **styles** `IStyleFunction<IMarqueeSelectionStyleProps, IMarqueeSelectionStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

# This is an advanced demo showing how you can use virtually all features of Fluent UI
# by creating custom components in JS and rendering them with shiny.react.
# This example is a translation of the example in
# https://developer.microsoft.com/en-us/fluentui#/controls/web/marqueeselection.

# Script showing how to:
# 1. Use mergeStyles and themes from Fluent
# 2. Define custom components
# 3. Send results back to Shiny.

customComponent <- function(name, js) {
  dependency <- htmltools::htmlDependency(
    name = name,
    version = "0", # Not used.
    src = c(href = ""), # Not used.
    head = paste0("
      <script>
        (jsmodule.CustomComponents ??= {}).", name, " = (() => {", js, "})();
      </script>
    ")
```

```
  )
  function(...) shiny.react::reactElement(
    module = "CustomComponents",
    name = name,
    props = shiny.react::asProps(...),
    deps = dependency
  )
}

MarqueeSelectionExample <- customComponent("MarqueeSelectionExample", "
  const React = jsmodule['react'];
  const Fluent = jsmodule['@fluentui/react'];

  const theme = Fluent.getTheme();
  const styles = Fluent.mergeStyleSets({
    photoList: {
      display: 'inline-block',
      border: '1px solid ' + theme.palette.neutralTertiary,
      margin: 0,
      padding: 10,
      overflow: 'hidden',
      userSelect: 'none',
    },

    photoCell: {
      position: 'relative',
      display: 'inline-block',
      margin: 2,
      boxSizing: 'border-box',
      background: theme.palette.neutralLighter,
      lineHeight: 100,
      verticalAlign: 'middle',
      textAlign: 'center',
      selectors: {
        '&.is-selected': {
          background: theme.palette.themeLighter,
          border: '1px solid ' + theme.palette.themePrimary,
        },
      },
    },
    checkbox: {
      margin: '10px 0',
    },
  });

  const useForceUpdate = () => {
    const [, setIt] = React.useState(false);
    return () => setIt(it => !it);
  };

  return function(params) {
    const forceUpdate = useForceUpdate();
    const inputId = params['inputId'];
```

```
    const photos = params['photos'];

    if (window.selection === undefined) {
      window.selection = new Fluent.Selection({
        items: photos,
        onSelectionChanged: function() {
          Shiny.setInputValue(inputId, window.selection.getSelectedIndices());
          forceUpdate();
        }
      });
    }

    const items = photos.map((photo, index) => {
      return React.createElement(
        'div',
        {
          key: index,
          'data-is-focusable': true,
          className: Fluent.css(
            styles.photoCell,
            window.selection.isIndexSelected(index) && 'is-selected'
          ),
          'data-selection-index': index,
          style: { width: photo.width, height: photo.height }
        },
        index
      );
    });

    return React.createElement(
      Fluent.MarqueeSelection,
      { selection: window.selection, isEnabled: true },
      React.createElement('ul', { className: styles.photoList }, items)
    );
  };
")

ui <- function(id) {
  ns <- NS(id)
  tagList(
    textOutput(ns("marqueeResult")),
    Label("Drag a rectangle around the items below to select them"),
    reactOutput(ns("marqueeSelection"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns

    photos <- lapply(1:50, function(index) {
      randomWidth <- 50 + sample.int(150, 1)
      list(
```

```
          key = index,
          url = paste0('http://placehold.it/', randomWidth, 'x100'),
          width = randomWidth,
          height = 100
        )
      })

      output$marqueeResult <- renderText({
        paste("You have selected: ", paste(input$selectedIndices, collapse = ", "))
      })

      output$marqueeSelection <- renderReact({
        MarqueeSelectionExample(
          inputId = ns("selectedIndices"),
          photos = photos
        )
      })
    }
  )
  }
  if (interactive()) {
    shinyApp(ui("app"), function(input, output) server("app"))
  }
```

`MaskedTextField` *TextField*

Description

Text fields (`TextField`) give people a way to enter and edit text. They're used in forms, modal dialogs, tables, and other surfaces where text input is required.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
MaskedTextField(...)

TextField(...)

TextField.shinyInput(inputId, ..., value = defaultValue)

updateTextField.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **ariaLabel** `string`
  Aria label for the text field.
- **autoAdjustHeight** `boolean`
  For multiline text fields, whether or not to auto adjust text field height.
- **autoComplete** `string`
  Whether the input field should have autocomplete enabled. This tells the browser to display options based on earlier typed values. Common values are 'on' and 'off' but for all possible values see the following links: https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete#Values https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill
- **borderless** `boolean`
  Whether or not the text field is borderless.
- **className** `string`
  Optional class name that is added to the container of the component.
- **componentRef** `IRefObject<ITextField>`
  Optional callback to access the ITextField component. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultValue** `string`
  Default value of the text field. Only provide this if the text field is an uncontrolled component; otherwise, use the `value` property.
- **deferredValidationTime** `number`
  Text field will start to validate after users stop typing for `deferredValidationTime` milliseconds. Updates to this prop will not be respected.
- **description** `string`
  Description displayed below the text field to provide additional details about what text to enter.
- **disabled** `boolean`
  Disabled state of the text field.
- **errorMessage** `string | JSX.Element`
  Static error message displayed below the text field. Use `onGetErrorMessage` to dynamically change the error message displayed (if any) based on the current value. `errorMessage` and `onGetErrorMessage` are mutually exclusive (`errorMessage` takes precedence).
- **iconProps** `IIconProps`
  Props for an optional icon, displayed in the far right end of the text field.
- **inputClassName** `string`
  Optional class name that is added specifically to the input/textarea element.

- **label** `string`
  Label displayed above the text field (and read by screen readers).
- **mask** `string`
  Only used by MaskedTextField: The masking string that defines the mask's behavior. A backslash will escape any character. Special format characters are: '9': `[0-9]` 'a': `[a-zA-Z]` '*': `[a-zA-Z0-9]`
- **maskChar** `string`
  Only used by MaskedTextField: The character to show in place of unfilled characters of the mask.
- **maskFormat** `{ [key: string]: RegExp; }`
  Only used by MaskedTextField: An object defining the format characters and corresponding regexp values. Default format characters: { '9': /`[0-9]`/, 'a': /`[a-zA-Z]`/, '*': /`[a-zA-Z0-9]`/ }
- **multiline** `boolean`
  Whether or not the text field is a multiline text field.
- **onChange** `(event: React.FormEvent<HTMLInputElement | HTMLTextAreaElement>, newValue?: string) =>`
  Callback for when the input value changes. This is called on both `input` and `change` events. (In a later version, this will probably only be called for the `change` event.)
- **onGetErrorMessage** `(value: string) => string | JSX.Element | PromiseLike<string | JSX.Element> | un`
  Function used to determine whether the input value is valid and get an error message if not. Mutually exclusive with the static string `errorMessage` (it will take precedence over this).

When it returns `string | JSX.Element`: - If valid, it returns empty string. - If invalid, it returns the error message and the text field will show a red border and show an error message below the text field.

When it returns `Promise<string | JSX.Element>`: - The resolved value is displayed as the error message. - If rejected, the value is thrown away.

- **onNotifyValidationResult** `(errorMessage: string | JSX.Element, value: string | undefined) => void`
  Function called after validation completes.
- **onRenderDescription** `IRenderFunction<ITextFieldProps>`
  Custom renderer for the description.
- **onRenderLabel** `IRenderFunction<ITextFieldProps>`
  Custom renderer for the label. If you don't call defaultRender, ensure that you give your custom-rendered label an id and that you set the textfield's aria-labelledby prop to that id.
- **onRenderPrefix** `IRenderFunction<ITextFieldProps>`
  Custom render function for prefix.
- **onRenderSuffix** `IRenderFunction<ITextFieldProps>`
  Custom render function for suffix.
- **prefix** `string`
  Prefix displayed before the text field contents. This is not included in the value. Ensure a descriptive label is present to assist screen readers, as the value does not include the prefix.
- **readOnly** `boolean`
  If true, the text field is readonly.
- **resizable** `boolean`
  For multiline text fields, whether or not the field is resizable.

- **styles** `IStyleFunctionOrObject<ITextFieldStyleProps, ITextFieldStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **suffix** `string`
  Suffix displayed after the text field contents. This is not included in the value. Ensure a descriptive label is present to assist screen readers, as the value does not include the suffix.
- **theme** `ITheme`
  Theme (provided through customization).
- **underlined** `boolean`
  Whether or not the text field is underlined.
- **validateOnFocusIn** `boolean`
  Run validation when focus moves into the input, and **do not** validate on change.

(Unless this prop and/or `validateOnFocusOut` is set to true, validation will run on every change.)

- **validateOnFocusOut** `boolean`
  Run validation when focus moves out of the input, and **do not** validate on change.

(Unless this prop and/or `validateOnFocusIn` is set to true, validation will run on every change.)

- **validateOnLoad** `boolean`
  Whether validation should run when the input is initially rendered.
- **value** `string`
  Current value of the text field. Only provide this if the text field is a controlled component where you are maintaining its current state; otherwise, use the `defaultValue` property.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a multiline text field when long entries are expected.
- Don't place a text field in the middle of a sentence, because the sentence structure might not make sense in all languages. For example, "Remind me in `[textfield]` weeks" should instead read, "Remind me in this many weeks: `[textfield]`".
- Format the text field for the expected entry. For example, when someone needs to enter a phone number, use an input mask to indicate that three sets of digits should be entered.

Content:

- Include a short label above the text field to communicate what information should be entered. Don't use placeholder text instead of a label. Placeholder text poses a variety of accessibility issues (including possible problems with color/contrast, and people thinking the form input is already filled out).
- When part of a form, make it clear which fields are required vs. optional. If the input is required, add "(required)" to the label. Don't exclusively use "*" to indicate required inputs as it is often not read by screen readers. For example, "First name (required)".
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    TextField.shinyInput(ns("text")),
    textOutput(ns("textValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$textValue <- renderText({
      sprintf("Value: %s", input$text)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Using custom handler to convert input to uppercase
CustomComponents <- tags$script(HTML("(function() {
  const { InputAdapter } = jsmodule['@/shiny.react'];
  const { TextField } = jsmodule['@fluentui/react'];
  const CustomComponents = jsmodule['CustomComponents'] ??= {};

  CustomComponents.UpperCaseTextField = InputAdapter(TextField, (value, setValue) => ({
    value: value.toUpperCase(),
    onChange: (e, v) => setValue(v.toUpperCase()),
  }));
})();"))

UpperCaseTextField <- function(inputId, ..., value = "") {
  shiny.react::reactElement(
    module = "CustomComponents",
    name = "UpperCaseTextField",
    props = shiny.react::asProps(inputId = inputId, ..., value = value),
    deps = shinyFluentDependency()
  )
}

ui <- function(id) {
  ns <- NS(id)
```

```
  tagList(
    CustomComponents,
    UpperCaseTextField(ns("uppercase_text")),
    textOutput(ns("text"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$text <- renderText(input$uppercase_text)
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`MessageBar` *MessageBar*

Description

A banner (`MessageBar`) displays errors, warnings, or important information about an open app or file. For example, if a file failed to upload an error message bar should appear.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
MessageBar(...)

MessageBarButton(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **actions** `JSX.Element`
  The actions you want to show on the other side.
- **ariaLabel** `string`
  A description of the message bar for the benefit of screen readers.
- **className** `string`
  Additional CSS class(es) to apply to the MessageBar.

- **componentRef** `IRefObject<IMessageBar>`
  Optional callback to access the IMessageBar interface. Use this instead of ref for accessing the public methods and properties of the component.
- **dismissButtonAriaLabel** `string`
  Aria label on dismiss button if onDismiss is defined.
- **dismissIconProps** `IIconProps`
  Custom icon prop to replace the dismiss icon. If unset, default will be the Fabric Clear icon.
- **isMultiline** `boolean`
  Determines if the message bar is multi lined. If false, and the text overflows over buttons or to another line, it is clipped.
- **messageBarIconProps** `IIconProps`
  Custom icon prop to replace the message bar icon. If unset, default will be the icon set by messageBarType.
- **messageBarType** `MessageBarType`
  The type of MessageBar to render.
- **onDismiss** `(ev?: React.MouseEvent<HTMLElement | BaseButton | Button>) => any`
  Whether the message bar has a dismiss button and its callback. If null, we don't show a dismiss button.
- **overflowButtonAriaLabel** `string`
  Aria label on overflow button if truncated is defined.
- **styles** `IStyleFunctionOrObject<IMessageBarStyleProps, IMessageBarStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)
- **truncated** `boolean`
  Determines if the message bar text is truncated. If true, a button will render to toggle between a single line view and multiline view. This prop is for single line message bars with no buttons only in a limited space scenario.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- A message bar is most commonly found near the top of an app, underneath the app's main command bar. For example, the Microsoft Office message bar is positioned beneath the Ribbon, but above the document canvas.
- Multiple message bars can appear at a time, but a given scenario or related set of scenarios should aim to only show one message bar at a time. Message bars are rarely shown in direct response to an action; rather, they should be shown when there's something a person should know about the overall app or document.

- Use the icons options to indicate the message type: the Info icon for information messages; ShieldAlert icon for security-related messages; the Warning icon for non-blocking errors; ErrorBadge icon for critical errors; the Blocked icon for blocking messages; and the Completed icon for success messages.

Content:

Message bars should include:

Title:
Limit titles to 50 characters (including spaces) to leave room for text expansion when translated. People should be able to scan the title to determine the purpose of the message. Capitalize only the first word of the title and any proper nouns.

Body text:
Describe the information or error state concisely, ideally in a single sentence. Limit the message to fewer than 512 characters (including spaces) to leave room for text expansion when translated. Include end punctuation for complete sentences.

Action buttons (Optional):
Offer one to two action buttons to help people solve any errors they're receiving. Limit button text to fewer than 50 charactesr (including spaces) to leave room for translation. Action buttons can have any callback attached to them and should provide people with options to address the notification and dismiss the message bar.

Link (Optional):
Don't use buttons when a subtler link will suffice. Reserve the use of a button for when the MessageBar has a single "hero" action that is useful at that particular moment. Avoid using more than one button.

Close button:
Always offer a quick way for people to close a message bar, unless there is an issue that must be resolved immediately, such as an expired subscription.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  MessageBar("Message")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Modal` *Modal*

Description

Modals are temporary pop-ups that take focus from the page or app and require people to interact with them. Unlike a dialog box (`Dialog`), a modal should be used for hosting lengthy content, such as privacy statements or license agreements, or for asking people to perform complex or multiple actions, such as changing settings.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Modal(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **allowTouchBodyScroll** `boolean`
  Allow body scroll on content and overlay on touch devices. Changing after mounting has no effect.
- **className** `string`
  Optional class name to be added to the root class
- **componentRef** `IRefObject<IModal>`
  Optional callback to access the IDialog interface. Use this instead of ref for accessing the public methods and properties of the component.
- **containerClassName** `string`
  Optional override for container class
- **dragOptions** `IDragOptions`
  The options to make the modal draggable
- **enableAriaHiddenSiblings** `boolean`
  Puts aria-hidden=true on all non-ancestors of the current modal, for screen readers. This is an experimental feature that will be graduated to default behavior after testing. This flag will be removed with the next major release.
- **isBlocking** `boolean`
  Whether the dialog can be light dismissed by clicking outside the dialog (on the overlay).
- **isDarkOverlay** `boolean`
  Whether the overlay is dark themed.

- **isModeless** `boolean`
  Whether the dialog should be modeless (e.g. not dismiss when focusing/clicking outside of the dialog). if true: isBlocking is ignored, there will be no overlay (isDarkOverlay is ignored), isClickableOutsideFocusTrap is true, and forceFocusInsideTrap is false
- **isOpen** `boolean`
  Whether the dialog is displayed.
- **layerProps** `ILayerProps`
  Defines an optional set of props to be passed through to Layer
- **onDismiss** `(ev?: React.MouseEvent<HTMLButtonElement>) => any`
  A callback function for when the Modal is dismissed light dismiss, before the animation completes.
- **onDismissed** `() => any`
  A callback function which is called after the Modal is dismissed and the animation is complete.
- **onLayerDidMount** `() => void`
  A callback function for when the Modal content is mounted on the overlay layer
- **overlay** `IOverlayProps`
  Defines an optional set of props to be passed through to Overlay
- **scrollableContentClassName** `string`
  Optional override for scrollable content class
- **styles** `IStyleFunctionOrObject<IModalStyleProps, IModalStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **subtitleAriaId** `string`
  ARIA id for the subtitle of the Modal, if any
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **titleAriaId** `string`
  ARIA id for the title of the Modal, if any
- **topOffsetFixed** `boolean`
  Whether the modal should have top offset fixed once opened and expand from the bottom only when the content changes dynamically.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use a variable width with a minimum width of 288 pixels.
- Use a variable height with a minimum height of 172 pixels.
- Center vertically and horizontally in the available space.
- Always have at least one focusable element inside a modal.
- Blocking modals (`Modeless Modal`) should be used very sparingly, only when it's critical for people to make a choice or provide information before they can proceed.

- Provide a clear way for people to dismiss the control, such as a Close button, which should always go in the upper right corner.

Content:

- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  tagList(
    reactOutput(ns("modal")),
    PrimaryButton.shinyInput(ns("showModal"), text = "Show modal"),
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    modalVisible <- reactiveVal(FALSE)
    observeEvent(input$showModal, modalVisible(TRUE))
    observeEvent(input$hideModal, modalVisible(FALSE))
    output$modal <- renderReact({
      Modal(isOpen = modalVisible(),
        Stack(tokens = list(padding = "15px", childrenGap = "10px"),
          div(style = list(display = "flex"),
            Text("Title", variant = "large"),
            div(style = list(flexGrow = 1)),
            IconButton.shinyInput(
              ns("hideModal"),
              iconProps = list(iconName = "Cancel")
            ),
          ),
          div(
            p("A paragraph of text."),
            p("Another paragraph.")
          )
        )
      )
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Nav` *Nav*

Description

Navs (also called "left nav" or "navigation pane") provide links to the main areas of an app or a site. In larger configurations, the Nav is always on-screen, usually on the left of the view. In smaller configurations, the Nav may collapse into a skinnier version or be completely hidden until the user taps an icon.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Nav(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **link** `INavLink`
  (Optional) Link to be rendered.
- **ariaLabel** `string`
  (Optional) The nav container aria label.
- **className** `string`
  Additional css class to apply to the Nav
- **componentRef** `IRefObject<INav>`
  Optional callback to access the INav interface. Use this instead of ref for accessing the public methods and properties of the component.
- **expandButtonAriaLabel** `string`
  (Optional) The nav container aria label. The link name is prepended to this label. If not provided, the aria label will default to the link name.
- **groups** `INavLinkGroup[] | null`
  A collection of link groups to display in the navigation bar
- **initialSelectedKey** `string`
  (Optional) The key of the nav item initially selected.
- **isOnTop** `boolean`
  Indicates whether the navigation component renders on top of other content in the UI
- **linkAs** `IComponentAs<INavButtonProps>`
  Render a custom link in place of the normal one. This replaces the entire button rather than simply button content

- **onLinkClick** `(ev?: React.MouseEvent<HTMLElement>, item?: INavLink) => void`
  Function callback invoked when a link in the navigation is clicked
- **onLinkExpandClick** `(ev?: React.MouseEvent<HTMLElement>, item?: INavLink) => void`
  Function callback invoked when the chevron on a link is clicked
- **onRenderGroupHeader** `IRenderFunction<IRenderGroupHeaderProps>`
  Used to customize how content inside the group header is rendered
- **onRenderLink** `IRenderFunction<INavLink>`
  Used to customize how content inside the link tag is rendered
- **selectedAriaLabel** `string`
  (Deprecated) Use ariaCurrent on links instead
- **selectedKey** `string`
  (Optional) The key of the nav item selected by caller.
- **styles** `IStyleFunctionOrObject<INavStyleProps, INavStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

navigation_styles <- list(
  root = list(
    height = "100%",
    boxSizing = "border-box",
    border = "1px solid #eee",
    overflowY = "auto"
  )
)

link_groups <- list(
  list(
    links = list(
      list(
        name = "Home",
        expandAriaLabel = "Expand Home section",
        collapseAriaLabel = "Collapse Home section",
        links = list(
          list(
            name = "Activity",
            url = "http://msn.com",
            key = "key1",
            target = "_blank"
```

```
            ),
            list(
              name = "MSN",
              url = "http://msn.com",
              disabled = TRUE,
              key = "key2",
              target = "_blank"
            )
          ),
          isExpanded = TRUE
        ),
        list(
          name = "Documents",
          url = "http://example.com",
          key = "key3",
          isExpanded = TRUE
        ),
        list(
          name = "Pages",
          url = "http://msn.com",
          key = "key4"
        ),
        list(
          name = "Notebook",
          url = "http://msn.com",
          key = "key5",
          disabled = TRUE
        ),
        list(
          name = "Communication and Media",
          url = "http://msn.com",
          key = "key6"
        ),
        list(
          name = "News",
          url = "http://cnn.com",
          icon = "News",
          key = "key7",
          target = "_blank",
          iconProps = list(
            iconName = "News",
            styles = list(
              root = list(
                fontSize = 20,
                color = "#106ebe"
              )
            )
          )
        )
      )
    )
  )
)
```

```
ui <- function(id) {
  ns <- NS(id)
  Nav(
    groups = link_groups,
    selectedKey = "key1",
    styles = navigation_styles
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
library(shiny.fluent)

# Custom rendering of group headers
navigation_styles <- list(
  root = list(
    height = "100%",
    width = "30%",
    boxSizing = "border-box",
    border = "1px solid #eee",
    overflowY = "auto"
  )
)

link_groups <- list(
  list(
    name = "Pages",
    links = list(
      list(name = "Activity"),
      list(name = "News")
    )
  ),
  list(
    name = "More Pages",
    links = list(
      list(name = "Settings"),
      list(name = "Notes")
    )
  )
)

ui <- function(id) {
  fluidPage(
    Nav(
      groups = link_groups,
```

```
      selectedKey = "key1",
      styles = navigation_styles,
      onRenderGroupHeader = JS("group => React.createElement('h3', null, group.name)")
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`OverflowSet` *OverflowSet*

Description

The OverflowSet is a flexible container component that is useful for displaying a primary set of content with additional content in an overflow callout. Note that the example below is only an example of how to render the component, not a specific use case.

Accessibility:

By default, the OverflowSet is simply `role=group`. If you used as a menu, you will need to add `role="menubar"` and add proper aria roles to each rendered item (`menuitem`, `menuitemcheckbox`, `menuitemradio`)

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
OverflowSet(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **key** `string`
  Unique id to identify the item.
- **keytipProps** `IKeytipProps`
  Optional keytip for the overflowSetItem.

- **className** `string`
  Class name
- **componentRef** `IRefObject<IOverflowSet>`
  Gets the component ref.
- **doNotContainWithinFocusZone** `boolean`
  If true do not contain the OverflowSet inside of a FocusZone, otherwise the OverflowSet will contain a FocusZone. If this is set to true focusZoneProps will be ignored. Use one or the other.
- **focusZoneProps** `IFocusZoneProps`
  Custom properties for OverflowSet's FocusZone. If doNotContainWithinFocusZone is set to true focusZoneProps will be ignored. Use one or the other.
- **items** `IOverflowSetItemProps[]`
  An array of items to be rendered by your onRenderItem function in the primary content area
- **itemSubMenuProvider** `(item: IOverflowSetItemProps) => any[] | undefined`
  Function that will take in an IOverflowSetItemProps and return the subMenu for that item. If not provided, will use 'item.subMenuProps.items' by default. This is only used if your overflow set has keytips.
- **keytipSequences** `string[]`
  Optional full keytip sequence for the overflow button, if it will have a keytip.
- **onRenderItem** `(item: IOverflowSetItemProps) => any`
  Method to call when trying to render an item.
- **onRenderOverflowButton** `IRenderFunction<any[]>`
  Rendering method for overflow button and contextual menu. The argument to the function is the overflowItems passed in as props to this function.
- **overflowItems** `IOverflowSetItemProps[]`
  An array of items to be passed to overflow contextual menu
- **overflowSide** `'start' | 'end'`
  Controls wether or not the overflow button is placed at the start or end of the items. This gives a reveresed visual behavior but maintains correct keyboard navigation.
- **role** `string`
  The role for the OverflowSet.
- **styles** `IStyleFunctionOrObject<IOverflowSetProps, IOverflowSetStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **vertical** `boolean`
  Change item layout direction to vertical/stacked. If role is set to `menubar`, `vertical={true}` will also add proper `aria-orientation`.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)
```

```
items <- list(
  list(key = "item1", icon = "Add", name = "Link 1"),
  list(key = "item2", icon = "Upload", name = "Link 2"),
  list(key = "item3", icon = "Share", name = "Link 3")
)
overflowItems <- list(
  list(key = "item4", icon = "Mail", name = "Overflow Link 1"),
  list(key = "item5", icon = "Calendar", name = "Overflow Link 2")
)
onRenderItem <- JS("item =>
  jsmodule['react'].createElement(jsmodule['@fluentui/react'].CommandBarButton, {
    role: 'menuitem',
    iconProps: { iconName: item.icon },
    styles: {
      root: { padding: '10px' }
    }
  })
")
onRenderOverflowButton <- JS("overflowItems =>
  jsmodule['react'].createElement(jsmodule['@fluentui/react'].CommandBarButton, {
    role: 'menuitem',
    title: 'More items',
    styles: {
      root: { padding: '10px' }
    },
    menuIconProps: { iconName: 'More' },
    menuProps: { items: overflowItems }
  })
")


ui <- function(id) {
  ns <- NS(id)
  OverflowSet(
    vertical = TRUE,
    items = items,
    overflowItems = overflowItems,
    onRenderItem = onRenderItem,
    onRenderOverflowButton = onRenderOverflowButton
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Overlay` *Overlay*

Description

Overlays are used to render a semi-transparent layer on top of existing UI. Overlays help focus the user on the content that sits above the added layer and are often used to help designate a modal or blocking experience. Overlays can be seen used in conjunction with Panels and Dialogs.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Overlay(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **allowTouchBodyScroll** `boolean`
  Allow body scroll on touch devices. Changing after mounting has no effect.
- **className** `string`
  Additional css class to apply to the Overlay
- **componentRef** `IRefObject<IOverlay>`
  Gets the component ref.
- **isDarkThemed** `boolean`
  Whether to use the dark-themed overlay.
- **onClick** `() => void`
- **styles** `IStyleFunctionOrObject<IOverlayStyleProps, IOverlayStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    DefaultButton.shinyInput(ns("toggleOverlay"), text = "Open Overlay"),
    reactOutput(ns("overlay"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    show <- reactiveVal(FALSE)
    observeEvent(input$toggleOverlay, show(!show()))
    output$overlay <- renderReact({
      if (show()) {
        Overlay(
          onClick = JS(paste0(
            "function() {",
            "  Shiny.setInputValue('", ns("toggleOverlay"), "', Math.random());",
            "}"
          )),
          isDarkThemed = TRUE,
          div(
            style = "background: white; width: 50vw; height: 20rem; margin: auto;",
            div(
              style = "padding: 2rem;",
              h1("Inside Overlay"),
              p("Click anywhere to hide.")
            )
          )
        )
      }
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Panel` *Panel*

Description

Panels are overlays that contain supplementary content and are used for complex creation, edit, or management experiences. For example, viewing details about an item in a list or editing settings.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Panel(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **allowTouchBodyScroll** `boolean`
  Allow body scroll on content and overlay on touch devices. Changing after mounting has no effect.
- **className** `string`
  Additional css class to apply to the Panel
- **closeButtonAriaLabel** `string`
  Aria label on close button
- **componentId** `string`
  Deprecated property. Serves no function.
- **componentRef** `IRefObject<IPanel>`
  Optional callback to access the IPanel interface. Use this instead of ref for accessing the public methods and properties of the component.
- **customWidth** `string`
  Custom panel width, used only when `type` is set to `PanelType.custom`.
- **elementToFocusOnDismiss** `HTMLElement`
  Sets the HTMLElement to focus on when exiting the FocusTrapZone.
- **firstFocusableSelector** `string`
  Indicates the selector for first focusable item. Deprecated, use `focusTrapZoneProps`.
- **focusTrapZoneProps** `IFocusTrapZoneProps`
  Optional props to pass to the FocusTrapZone component to manage focus in the panel.
- **forceFocusInsideTrap** `boolean`
  Indicates whether Panel should force focus inside the focus trap zone. If not explicitly specified, behavior aligns with FocusTrapZone's default behavior. Deprecated, use `focusTrapZoneProps`.
- **hasCloseButton** `boolean`
  Has the close button visible.
- **headerClassName** `string`
  Optional parameter to provider the class name for header text
- **headerText** `string`
  Header text for the Panel.
- **headerTextProps** `React.HTMLAttributes<HTMLDivElement>`
  The props for header text container.

- **ignoreExternalFocusing** `boolean`
  Indicates if this Panel will ignore keeping track of HTMLElement that activated the Zone. Deprecated, use `focusTrapZoneProps`.
- **isBlocking** `boolean`
  Whether the panel uses a modal overlay or not
- **isFooterAtBottom** `boolean`
  Determines if content should stretch to fill available space putting footer at the bottom of the page
- **isHiddenOnDismiss** `boolean`
  Whether the panel is hidden on dismiss, instead of destroyed in the DOM. Protects the contents from being destroyed when the panel is dismissed.
- **isLightDismiss** `boolean`
  Whether the panel can be light dismissed.
- **isOpen** `boolean`
  Whether the panel is displayed. If true, will cause panel to stay open even if dismissed. If false, will cause panel to stay hidden. If undefined, will allow the panel to control its own visility through open/dismiss methods.
- **layerProps** `ILayerProps`
  Optional props to pass to the Layer component hosting the panel.
- **onDismiss** `(ev?: React.SyntheticEvent<HTMLElement>) => void`
  A callback function for when the panel is closed, before the animation completes. If the panel should NOT be dismissed based on some keyboard event, then simply call ev.preventDefault() on it
- **onDismissed** `() => void`
  A callback function which is called **after** the Panel is dismissed and the animation is complete. (If you need to update the Panel's `isOpen` prop in response to a dismiss event, use `onDismiss` instead.)
- **onLightDismissClick** `() => void`
  Optional custom function to handle clicks outside the panel in lightdismiss mode
- **onOpen** `() => void`
  A callback function for when the Panel is opened, before the animation completes.
- **onOpened** `() => void`
  A callback function for when the Panel is opened, after the animation completes.
- **onOuterClick** `() => void`
  Optional custom function to handle clicks outside this component
- **onRenderBody** `IRenderFunction<IPanelProps>`
  Optional custom renderer for body region. Replaces any children passed into the component.
- **onRenderFooter** `IRenderFunction<IPanelProps>`
  Optional custom renderer for footer region. Replaces sticky footer.
- **onRenderFooterContent** `IRenderFunction<IPanelProps>`
  Custom renderer for content in the sticky footer
- **onRenderHeader** `IPanelHeaderRenderer`
  Optional custom renderer for header region. Replaces current title

- **onRenderNavigation** `IRenderFunction<IPanelProps>`
  Optional custom renderer navigation region. Replaces the region that contains the close button.
- **onRenderNavigationContent** `IRenderFunction<IPanelProps>`
  Optional custom renderer for content in the navigation region. Replaces current close button.
- **overlayProps** `IOverlayProps`
  Optional props to pass to the Overlay component that the panel uses.
- **styles** `IStyleFunctionOrObject<IPanelStyleProps, IPanelStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **type** `PanelType`
  Type of the panel.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use for self-contained experiences where someone doesn't need to interact with the app view to complete the task.
- Consider how the panel and its contained contents will scale across responsive web breakpoints.

Header:

- Include a close button in the top-right corner.
- Lock the title to the top of the panel.
- The header can contain a variety of components. Components are stacked under the main title, locked to the top, and push content down.

Body:

- The body is a flexible container that can house a wide variety of components, content, and formatting.
- Content can scroll.

Footer:

- Standard footers include primary and secondary buttons to confirm or cancel the changes or task.
- Read-only panels may contain a single Close button.
- Lock the footer to the bottom of the panel.

Content:

Title:

- Titles should explain the panel content in clear, concise, and specific terms.
- Keep the length of the title to one line, if possible.

- Use sentence-style capitalization—only capitalize the first word. For more info, see [Capitalization] in the Microsoft Writing Style Guide.
- Don't put a period at the end of the title.

[capitalization]: https://docs.microsoft.com/style-guide/capitalization

Button labels:

- Be concise. Limit labels to one or two words. Usually a single verb is best. Include a noun if there's any room for interpretation about what the verb means. For example, "Save" or "Save settings."

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    DefaultButton.shinyInput(ns("showPanel"), text = "Open panel"),
    reactOutput(ns("reactPanel"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    isPanelOpen <- reactiveVal(FALSE)
    output$reactPanel <- renderReact({
      Panel(
        headerText = "Sample panel",
        isOpen = isPanelOpen(),
        "Content goes here.",
        onDismiss = JS(paste0(
          "function() {",
          "  Shiny.setInputValue('", ns("hidePanel"), "', Math.random());",
          "}"
        ))
      )
    })
    observeEvent(input$showPanel, isPanelOpen(TRUE))
    observeEvent(input$hidePanel, isPanelOpen(FALSE))
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

parseTheme *parseTheme*

Description

Reads a theme JSON generated by Theme Designer: https://fabricweb.z5.web.core.windows.net/pr-deploy-site/refs/heads/master/theming-designer/ and parses it to an object digestable by ThemeProvider

Usage

```
parseTheme(path = NULL, json = NULL)
```

Arguments

path	A path to JSON file containing the theme created in Theme Designer
json	A JSON string containing the theme created in Theme Designer

Value

A list with Fluent theme that can be used in ThemeProvider

See Also

ThemeProvider() for usage of this function

Persona *Persona*

Description

A persona is a visual representation of a person across products, typically showcasing the image that person has chosen to upload themselves. The control can also be used to show that person's online status.

The complete control inclues an individual's avatar (an uploaded image or a composition of the person's initials on a background color), their name or identification, and online status.

The persona control is used in the PeoplePicker and Facepile controls.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Persona(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the PersonaCoin
- **componentRef** `IRefObject<{}>`
  Gets the component ref.
- **styles** `IStyleFunctionOrObject<IPersonaCoinStyleProps, IPersonaCoinStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **componentRef** `IRefObject<{}>`
  Gets the component ref.
- **styles** `IStyleFunctionOrObject<IPersonaPresenceStyleProps, IPersonaPresenceStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **className** `string`
  Additional CSS class(es) to apply to the Persona
- **componentRef** `IRefObject<IPersona>`
  Optional callback to access the IPersona interface. Use this instead of ref for accessing the public methods and properties of the component.
- **onRenderOptionalText** `IRenderFunction<IPersonaProps>`
  Optional custom renderer for the optional text.
- **onRenderPrimaryText** `IRenderFunction<IPersonaProps>`
  Optional custom renderer for the primary text.
- **onRenderSecondaryText** `IRenderFunction<IPersonaProps>`
  Optional custom renderer for the secondary text.
- **onRenderTertiaryText** `IRenderFunction<IPersonaProps>`
  Optional custom renderer for the tertiary text.
- **styles** `IStyleFunctionOrObject<IPersonaStyleProps, IPersonaStyles>`
  Call to provide customized styling that will layer on top of variant rules
- **allowPhoneInitials** `boolean`
  Whether initials are calculated for phone numbers and number sequences. Example: Set property to true to get initials for project names consisting of numbers only.
- **coinProps** `IPersonaCoinProps`
  Optional HTML element props for Persona coin.
- **coinSize** `number`
  Optional custom persona coin size in pixel.
- **hidePersonaDetails** `boolean`
  Whether to not render persona details, and just render the persona image/initials.
- **imageAlt** `string`
  Alt text for the image to use. Defaults to an empty string.

- **imageInitials** `string`
  The user's initials to display in the image area when there is no image.
- **imageShouldFadeIn** `boolean`
  If true, adds the css class 'is-fadeIn' to the image.
- **imageShouldStartVisible** `boolean`
  If true, the image starts as visible and is hidden on error. Otherwise, the image is hidden until it is successfully loaded. This disables imageShouldFadeIn.
- **imageUrl** `string`
  Url to the image to use, should be a square aspect ratio and big enough to fit in the image area.
- **initialsColor** `PersonaInitialsColor | string`
  The background color when the user's initials are displayed.
- **isOutOfOffice** `boolean`
  This flag can be used to signal the persona is out of office. This will change the way the presence icon looks for statuses that support dual-presence.
- **onPhotoLoadingStateChange** `(newImageLoadState: ImageLoadState) => void`
  Optional callback for when loading state of the photo changes
- **onRenderCoin** `IRenderFunction<IPersonaSharedProps>`
  Optional custom renderer for the coin
- **onRenderInitials** `IRenderFunction<IPersonaSharedProps>`
  Optional custom renderer for the initials
- **onRenderPersonaCoin** `IRenderFunction<IPersonaSharedProps>`
  Optional custom renderer for the coin
- **optionalText** `string`
  Optional text to display, usually a custom message set. The optional text will only be shown when using size100.
- **presence** `PersonaPresence`
  Presence of the person to display - will not display presence if undefined.
- **presenceColors** `{ available: string; away: string; busy: string; dnd: string; offline: string; oof: string; background: string; }`
  The colors to be used for the presence-icon and it's background
- **presenceTitle** `string`
  Presence title to be shown as a tooltip on hover over the presence icon.
- **primaryText** `string`
  Primary text to display, usually the name of the person.
- **secondaryText** `string`
  Secondary text to display, usually the role of the user.
- **showInitialsUntilImageLoads** `boolean`
  If true renders the initials while the image is loading. This only applies when an imageUrl is provided.
- **showSecondaryText** `boolean`
- **showUnknownPersonaCoin** `boolean`
  If true, show the special coin for unknown persona. It has '?' in place of initials, with static font and background colors

- **size** `PersonaSize`
  Decides the size of the control.
- **tertiaryText** `string`
  Tertiary text to display, usually the status of the user. The tertiary text will only be shown when using size72 or size100.
- **text** `string`
  Primary text to display, usually the name of the person.
- **theme** `ITheme`
  Theme provided by High-Order Component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use the 24-pixel persona in text fields in read-only mode or in experiences like multicolumn lists which need compact persona representations.
- Use the 32-pixel persona in text fields in edit mode.
- Use the 32-pixel, 40-pixel, and 48-pixel persona in menus and list views.
- Use the 72-pixel and 100-pixel persona in profile cards and views.

Content:

- Change the values of the color swatches in high contrast mode.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Persona(
    imageInitials = "AL",
    text = "Annie Lindqvist",
    secondaryText = "Software Engineer",
    presence = 4
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Pivot` *Pivot*

Description

The Pivot control and related tabs pattern are used for navigating frequently accessed, distinct content categories. Pivots allow for navigation between two or more content views and relies on text headers to articulate the different sections of content.

- Tapping on a pivot item header navigates to that header's section content.

Tabs are a visual variant of Pivot that use a combination of icons and text or just icons to articulate section content.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Pivot(...)

PivotItem(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **alwaysRender** `boolean`
  Defines whether to always render the pivot item (regardless of whether it is selected or not). Useful if you're rendering content that is expensive to mount.
- **ariaLabel** `string`
  The aria label of each pivot link which will read by screen reader instead of linkText.

Note that unless you have compelling requirements you should not override aria-label.

- **componentRef** `IRefObject<{}>`
  Gets the component ref.
- **headerButtonProps** `IButtonProps | { [key: string]: string | number | boolean; }`
  Props for the header command button. This provides a way to pass in native props, such as data-* and aria-*, for each pivot header/link element.
- **headerText** `string`
  The text displayed of each pivot link.
- **itemCount** `number | string`
  Defines an optional item count displayed in parentheses just after the `linkText`.

Examples: completed (4), Unread (99+)

- **itemIcon** `string`
  An optional icon to show next to the pivot link.
- **itemKey** `string`
  An required key to uniquely identify a pivot item.

Note: The 'key' from react props cannot be used inside component.

- **keytipProps** `IKeytipProps`
  Optional keytip for this PivotItem.
- **linkText** `string`
  The text displayed of each pivot link - renaming to `headerText`.
- **onRenderItemLink** `IRenderFunction<IPivotItemProps>`
  Optional custom renderer for the pivot item link.
- **className** `string`
  Additional css class to apply to the Pivot
- **componentRef** `IRefObject<IPivot>`
  Optional callback to access the IPivot interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultSelectedIndex** `number`
  Default selected index for the pivot. Only provide this if the pivot is an uncontrolled component; otherwise, use the `selectedKey` property.

This property is also mutually exclusive with `defaultSelectedKey`.

- **defaultSelectedKey** `string`
  Default selected key for the pivot. Only provide this if the pivot is an uncontrolled component; otherwise, use the `selectedKey` property.

This property is also mutually exclusive with `defaultSelectedIndex`.

- **getTabId** `(itemKey: string, index: number) => string`
  Callback to customize how IDs are generated for each tab header. Useful if you're rendering content outside and need to connect aria-labelledby.
- **headersOnly** `boolean`
  Whether to skip rendering the tabpanel with the content of the selected tab. Use this prop if you plan to separately render the tab content and don't want to leave an empty tabpanel in the page that may confuse Screen Readers.
- **initialSelectedIndex** `number`
  Index of the pivot item initially selected. Mutually exclusive with `initialSelectedKey`. Only provide this if the pivot is an uncontrolled component; otherwise, use `selectedKey`.
- **initialSelectedKey** `string`
  Key of the pivot item initially selected. Mutually exclusive with `initialSelectedIndex`. Only provide this if the pivot is an uncontrolled component; otherwise, use `selectedKey`.
- **linkFormat** `PivotLinkFormat`
  PivotLinkFormat to use (links, tabs)

- **linkSize** `PivotLinkSize`
  PivotLinkSize to use (normal, large)
- **onLinkClick** `(item?: PivotItem, ev?: React.MouseEvent<HTMLElement>) => void`
  Callback for when the selected pivot item is changed.
- **selectedKey** `string | null`
  Key of the selected pivot item. Updating this will override the Pivot's selected item state. Only provide this if the pivot is a controlled component where you are maintaining the current state; otherwise, use `defaultSelectedKey`.
- **styles** `IStyleFunctionOrObject<IPivotStyleProps, IPivotStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Pivot(
    PivotItem(headerText = "Tab 1", Label("Hello 1")),
    PivotItem(headerText = "Tab 2", Label("Hello 2"))
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

ProgressIndicator *ProgressIndicator*

Description

ProgressIndicators are used to show the completion status of an operation lasting more than 2 seconds. If the state of progress cannot be determined, use a Spinner instead. ProgressIndicators can appear in a new panel, a flyout, under the UI initiating the operation, or even replacing the initiating UI, as long as the UI can return if the operation is canceled or is stopped.

ProgressIndicators feature a bar showing total units to completion, and total units finished. The description of the operation appears above the bar, and the status in text appears below. The description should tell someone exactly what the operation is doing. Examples of formatting include:

- `[Object]` is being `[operation name]`, or
- `[Object]` is being `[operation name]` to `[destination name]` or
- `[Object]` is being `[operation name]` from `[source name]` to `[destination name]`

Status text is generally in units elapsed and total units. If the operation can be canceled, an "X" icon is used and should be placed in the upper right, aligned with the baseline of the operation name. When an error occurs, replace the status text with the error description using ms-fontColor-redDark.

Real-world examples include copying files to a storage location, saving edits to a file, and more. Use units that are informative and relevant to give the best idea to users of how long the operation will take to complete. Avoid time units as they are rarely accurate enough to be trustworthy. Also, combine steps of a complex operation into one total bar to avoid "rewinding" the bar. Instead change the operation description to reflect the change if necessary. Bars moving backwards reduce confidence in the service.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ProgressIndicator(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaValueText** `string`
  Text alternative of the progress status, used by screen readers for reading the value of the progress.
- **barHeight** `number`
  Height of the ProgressIndicator
- **className** `string`
  Additional css class to apply to the ProgressIndicator
- **description** `React.ReactNode`
  Text describing or supplementing the operation. May be a string or React virtual elements.
- **label** `React.ReactNode`
  Label to display above the control. May be a string or React virtual elements.
- **onRenderProgress** `IRenderFunction<IProgressIndicatorProps>`
  A render override for the progress track.
- **percentComplete** `number`
  Percentage of the operation's completeness, numerically between 0 and 1. If this is not set, the indeterminate progress animation will be shown instead.

- **progressHidden** `boolean`
  Whether or not to hide the progress state.
- **styles** `IStyleFunctionOrObject<IProgressIndicatorStyleProps, IProgressIndicatorStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **title** `string`
  Deprecated at v0.43.0, to be removed at \>= v0.53.0. Use `label` instead.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  ProgressIndicator(
    label = "Example title",
    description = "Example description"
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Rating` *Rating*

Description

Ratings show people's opinions of a product, helping others make more informed purchasing decisions. People can also rate products they've purchased.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Rating(...)

Rating.shinyInput(inputId, ..., value = defaultValue)

updateRating.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **allowZeroStars** `boolean`
  Allow the rating value to be set to 0 instead of a minimum of 1.
- **ariaLabelFormat** `string`
  Optional label format for a rating star that will be read by screen readers. Can be used like "{0} of {1} stars selected", where {0} will be substituted by the current rating and {1} will be substituted by the max rating.
- **ariaLabelId** `string`
  Deprecated: Optional id of label describing this instance of Rating.
- **componentRef** `IRefObject<IRating>`
  Optional callback to access the IRating interface. Use this instead of ref for accessing the public methods and properties of the component.
- **getAriaLabel** `(rating: number, max: number) => string`
- **icon** `string`
  Custom icon
- **max** `number`
  Maximum rating, defaults to 5, has to be \>= min
- **min** `number`
  Minimum rating, defaults to 1, has to be \>= 0
- **onChange** `(event: React.FocusEvent<HTMLElement>, rating?: number) => void`
  Callback issued when the rating changes.
- **onChanged** `(rating: number) => void`

- **rating** `number`
  Selected rating, has to be an integer between min and max
- **readOnly** `boolean`
  Optional flag to mark rating control as readOnly
- **size** `RatingSize`
  Size of rating, defaults to small
- **styles** `IStyleFunctionOrObject<IRatingStyleProps, IRatingStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)
- **unselectedIcon** `string`
  Custom icon for unselected rating elements.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Make it clear which item the rating pertains to by making sure the layout and grouping are clear when several items are on the page.
- Don't use the rating component for data that has a continuous range, such as the brightness of a photo. Instead, use a slider.

Content:

- Use a five-star rating system.
- Use sentence-style capitalization—only capitalize the first word. For more info, see Capitalization in the Microsoft Writing Style Guide.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    Rating.shinyInput(ns("rating"), value = 2),
    textOutput(ns("ratingValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$ratingValue <- renderText({
      sprintf("Value: %s", input$rating)
```

```
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`ResizeGroup` *ResizeGroup*

Description

ResizeGroup is a React component that can be used to help fit the right amount of content within a container. The consumer of the ResizeGroup provides some initial data, a reduce function, and a render function. The render function is responsible for populating the contents of a the container when given some data. The initial data should represent the data that should be rendered when the ResizeGroup has infinite width. If the contents returned by the render function do not fit within the ResizeGroup, the reduce function is called to get a version of the data whose rendered width should be smaller than the data that was just rendered.

An example scenario is shown below, where controls that do not fit on screen are rendered in an overflow menu. The data in this situation is a list of 'primary' controls that are rendered on the top level and a set of overflow controls rendered in the overflow menu. The initial data in this case has all the controls in the primary set. The implementation of onReduceData moves a control from the overflow well into the primary control set.

This component queries the DOM for the dimensions of elements. Too many of these dimension queries will negatively affect the performance of the component and could cause poor interactive performance on websites. One way to reduce the number of measurements performed by the component is to provide a cacheKey in the initial data and in the return value of onReduceData. Two data objects with the same cacheKey are assumed to have the same width, resulting in measurements being skipped for that data object. In the controls with an overflow example, the cacheKey is simply the concatenation of the keys of the controls that appear in the top level.

There is a boolean context property (isMeasured) added to let child components know if they are only being used for measurement purposes. When isMeasured is true, it will signify that the component is not the instance visible to the user. This will not be needed for most scenarios. It is intended to be used to to skip unwanted side effects of mounting a child component more than once. This includes cases where the component makes API requests, requests focus to one of its elements, expensive computations, and/or renders markup unrelated to its size. Be careful not to use this property to change the components rendered output in a way that effects it size in any way.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ResizeGroup(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the Component
- **componentRef** `IRefObject<IResizeGroup>`
  Optional callback to access the IResizeGroup interface. Use this instead of ref for accessing the public methods and properties of the component.
- **data** `any`
  Initial data to be passed to the `onRenderData` function. When there is no `onGrowData` provided, this data should represent what should be passed to the render function when the parent container of the ResizeGroup is at its maximum supported width. A `cacheKey` property may optionally be included as part of the data. Two data objects with the same `cacheKey` will be assumed to take up the same width and will prevent measurements. The type of `cacheKey` is a string.
- **dataDidRender** `(renderedData: any) => void`
  Function to be called every time data is rendered. It provides the data that was actually rendered. A use case would be adding telemetry when a particular control is shown in an overflow well or dropped as a result of onReduceData or to count the number of renders that an implementation of onReduceData triggers.
- **direction** `ResizeGroupDirection`
  Direction of this resize group, vertical or horizontal
- **onGrowData** `(prevData: any) => any`
  Function to be performed on the data in order to increase its width. It is called in scenarios where the container has more room than the previous render and we may be able to fit more content. If there are no more scaling operations to perform on teh data, it should return undefined to prevent an infinite render loop.
- **onReduceData** `(prevData: any) => any`
  Function to be performed on the data in order to reduce its width and make it fit into the given space. If there are no more scaling steps to apply, it should return undefined to prevent an infinite render loop.
- **onRenderData** `(data: any) => JSX.Element`
  Function to render the data. Called when rendering the contents to the screen and when rendering in a hidden div to measure the size of the contents.
- **styles** `IStyleFunctionOrObject<IResizeGroupStyleProps, IResizeGroupStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

data <- list(
  items = list(
    "many", "strings", "with", "varying", "length", "sometimes", "very", "short",
    "other", "times", "extraordinarily", "long"
  )
)
onRenderData <- JS("data =>
  data.items.map(item =>
    jsmodule['react'].createElement('div',
      {
        style: {
          display: 'inline-block',
          backgroundColor: 'orange',
          padding: '10px',
          margin: '10px',
          fontSize: '20px',
        }
      },
      item
    )
  )
")
onReduceData <- JS("data => ({ items: data.items.slice(0, -1) })")

ui <- function(id) {
  ns <- NS(id)
  div(
    p("Resize the browser to see how the elements are hidden when they do not fit:"),
    ResizeGroup(
      data = data,
      onRenderData = onRenderData,
      onReduceData = onReduceData
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

---

runExample *Run example shiny.fluent apps.*

Description

Based on shiny::runExample, and takes the same arguments.

Usage

```
runExample(
  example = NA,
  port = getOption("shiny.port"),
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"),
  display.mode = c("auto", "normal", "showcase")
)
```

Arguments

example	Example to run. NA to list the examples.
port	The TCP port that the application should listen on
launch.browser	Whether to open the app in a browser
host	The IPv4 address to listen on.
display.mode	Display mode for the app.

Value

This function normally does not return; interrupt R to stop the application (usually by pressing Ctrl+C or Esc).

ScrollablePane *ScrollablePane*

Description

A scrollable pane (ScrollablePane) is a helper component that's used with the Sticky component. It will "stick" to the top or bottom of the scrollable region and remain visible.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
ScrollablePane(...)

Sticky(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **className** `string`
  Additional css class to apply to the ScrollablePane
- **componentRef** `IRefObject<IScrollablePane>`
  Optional callback to access the IScrollablePane interface. Use this instead of ref for accessing the public methods and properties of the component.
- **initialScrollPosition** `number`
  Sets the initial scroll position of the ScrollablePane
- **scrollbarVisibility** `ScrollbarVisibility`
- **styles** `IStyleFunctionOrObject<IScrollablePaneStyleProps, IScrollablePaneStyles>`
  Call to provide customized styling that will layer on top of the variant rules
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Use the sticky component on block-level elements.
- Sticky components should ideally be section headers and/or footers.
- Use `position: absolute`. Ensure that the parent element has an explicit `height` and `position: relative`, or has space already allocated for the scrollable pane.
- Ensure that the total height of `Sticky` components does not exceed the height of the `ScrollablePane`.

Examples

```
library(shiny)
library(shiny.fluent)

pane <- function(header, paragraphs) (
  div(
    Sticky(
      div(
      style = "background-color: #80CAF1; border-top: 1px solid; border-bottom: 1px solid",
        header
      )
    ),
    stringi::stri_rand_lipsum(paragraphs)
  )
)

ui <- function(id) {
  ns <- NS(id)
  ScrollablePane(
```

```
    styles = list(
      root = list(position = "relative", height = "500px", width = "400px")
    ),
    pane("Some text", 3),
    pane("A lot of text", 5),
    pane("Just a short ending", 1)
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

SearchBox *SearchBox*

Description

A search box (SearchBox) provides an input field for searching content within a site or app to find specific items.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
SearchBox(...)

SearchBox.shinyInput(inputId, ..., value = defaultValue)

updateSearchBox.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **ariaLabel** `string`
  The aria label of the SearchBox for the benefit of screen readers.
- **className** `string`
  CSS class to apply to the SearchBox.
- **clearButtonProps** `IButtonProps`
  The props for the clear button.
- **componentRef** `IRefObject<ISearchBox>`
  Optional callback to access the ISearchBox interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultValue** `string`
  The default value of the text in the SearchBox, in the case of an uncontrolled component. This prop is being deprecated since so far, uncontrolled behavior has not been implemented.
- **disableAnimation** `boolean`
  Whether or not to animate the SearchBox icon on focus.
- **iconProps** `Pick<IIconProps, Exclude<keyof IIconProps, 'className'>>`
  The props for the icon.
- **labelText** `string`
  Deprecated. Use `placeholder` instead.
- **onChange** `(event?: React.ChangeEvent<HTMLInputElement>, newValue?: string) => void`
  Callback function for when the typed input for the SearchBox has changed.
- **onChanged** `(newValue: any) => void`
  Deprecated at v0.52.2, use `onChange` instead.
- **onClear** `(ev?: any) => void`
  Callback executed when the user clears the search box by either clicking 'X' or hitting escape.
- **onEscape** `(ev?: any) => void`
  Callback executed when the user presses escape in the search box.
- **onSearch** `(newValue: any) => void`
  Callback executed when the user presses enter in the search box.
- **placeholder** `string`
  Placeholder for the search box.
- **styles** `IStyleFunctionOrObject<ISearchBoxStyleProps, ISearchBoxStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization).
- **underlined** `boolean`
  Whether or not the SearchBox is underlined.
- **value** `string`
  The value of the text in the SearchBox.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Don't build a custom search control based on the default text box or any other control.
- Use a search box without a parent container when it's not restricted to a certain width to accommodate other content. This search box will span the entire width of the space it's in.

Content:

- Use placeholder text in the search box to describe what people can search for. For example, "Search", "Search files", or "Search contacts list".
- Although search entry points tend to be similarly visualized, they can provide access to results that range from broad to narrow. By effectively communicating the scope of a search, you can ensure that people's expectations are met by the capabilities of the search you're performing, which will reduce the possibility of frustration. The search entry point should be placed near the content being searched.Some common search scopes include:
  - **Global**: Search across multiple sources of cloud and local content. Varied results include URLs, documents, media, actions, apps, and more.
  - **Web**: Search a web index. Results include pages, entities, and answers.
  - **My stuff**: Search across devices, cloud, social graphs, and more. Results are varied but are constrained by the connection to user accounts.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    SearchBox.shinyInput(ns("search"), placeholder = "Search"),
    textOutput(ns("searchValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$searchValue <- renderText({
      sprintf("Value: %s", input$search)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Separator` *Separator*

Description

A separator visually separates content into groups.

You can render content in the separator by specifying the component's children. The component's children can be plain text or a component like Icon. The content is center-aligned by default.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Separator(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **alignContent** `'start' | 'center' | 'end'`
  Where the content should be aligned in the separator.
- **styles** `IStyleFunctionOrObject<ISeparatorStyleProps, ISeparatorStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)
- **vertical** `boolean`
  Whether the element is a vertical separator.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Separator("Text")
}

server <- function(id) {
```

```
  moduleServer(id, function(input, output, session) {})
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Shimmer` *Shimmer*

Description

Shimmer is a temporary animation placeholder for when a service call takes time to return data and we don't want to block rendering the rest of the UI.

If a smooth transition from Shimmer to content is desired, wrap the content node with a Shimmer element and use the `isDataLoaded` prop to trigger the transition. In cases where the content node is not wrapped in a Shimmer, use the `shimmerElements` or `customElementsGroup` props, and once data arrives, manually replace the Shimmer UI with the intended content. See the examples below for reference.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Shimmer(...)

ShimmerElementsGroup(...)

ShimmeredDetailsList(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **borderStyle** `IRawStyle`
  Use to set custom styling of the shimmerCircle borders.
- **componentRef** `IRefObject<IShimmerCircle>`
  Optional callback to access the IShimmerCircle interface. Use this instead of ref for accessing the public methods and properties of the component.
- **height** `number`
  Sets the height of the circle.
- **styles** `IStyleFunctionOrObject<IShimmerCircleStyleProps, IShimmerCircleStyles>`
  Call to provide customized styling that will layer on top of the variant rules.

- **theme** `ITheme`
  Theme provided by High-Order Component.
- **backgroundColor** `string`
  Defines the background color of the space in between and around shimmer elements.
- **componentRef** `IRefObject<IShimmerElementsGroup>`
  Optional callback to access the IShimmerElementsGroup interface. Use this instead of ref for accessing the public methods and properties of the component.
- **flexWrap** `boolean`
  Optional boolean for enabling flexWrap of the container containing the shimmerElements.
- **rowHeight** `number`
  Optional maximum row height of the shimmerElements container.
- **shimmerElements** `IShimmerElement[]`
  Elements to render in one group of the Shimmer.
- **styles** `IStyleFunctionOrObject<IShimmerElementsGroupStyleProps, IShimmerElementsGroupStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **width** `string`
  Optional width for ShimmerElements container.
- **borderStyle** `IRawStyle`
  Use to set custom styling of the shimmerGap borders.
- **componentRef** `IRefObject<IShimmerGap>`
  Optional callback to access the IShimmerGap interface. Use this instead of ref for accessing the public methods and properties of the component.
- **height** `number`
  Sets the height of the gap.
- **styles** `IStyleFunctionOrObject<IShimmerGapStyleProps, IShimmerGapStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **width** `number | string`
  Sets width value of the gap.
- **borderStyle** `IRawStyle`
  Use to set custom styling of the shimmerLine borders.
- **componentRef** `IRefObject<IShimmerLine>`
  Optional callback to access the IShimmerLine interface. Use this instead of ref for accessing the public methods and properties of the component.
- **height** `number`
  Sets the height of the rectangle.
- **styles** `IStyleFunctionOrObject<IShimmerLineStyleProps, IShimmerLineStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.

- **width** `number | string`
  Sets width value of the line.
- **ariaLabel** `string`
  Localized string of the status label for screen reader
- **className** `string`
  Additional CSS class(es) to apply to the Shimmer container.
- **componentRef** `IRefObject<IShimmer>`
  Optional callback to access the IShimmer interface. Use this instead of ref for accessing the public methods and properties of the component.
- **customElementsGroup** `React.ReactNode`
  Custom elements when necessary to build complex placeholder skeletons.
- **isDataLoaded** `boolean`
  Controls when the shimmer is swapped with actual data through an animated transition.
- **shimmerColors** `IShimmerColors`
  Defines an object with possible colors to pass for Shimmer customization used on different backgrounds.
- **shimmerElements** `IShimmerElement[]`
  Elements to render in one line of the Shimmer.
- **styles** `IStyleFunctionOrObject<IShimmerStyleProps, IShimmerStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **width** `number | string`
  Sets the width value of the shimmer wave wrapper.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  tagList(
    div(
     p("Basic Shimmer with no elements provided. It defaults to a line of 16px height."),
      Shimmer(),
      Shimmer(width = "75%"),
      Shimmer(width = "50%")
    ),
    tags$head(tags$style(
      ".ms-Shimmer-container { margin: 10px 0 }"
    ))
  )
```

```
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`shinyFluentDependency` *Shiny Fluent JS dependency*

Description

Shiny Fluent JS dependency

Usage

```
shinyFluentDependency()
```

Value

HTML dependency object.

`Slider` *Slider*

Description

A slider provides a visual indication of adjustable content, as well as the current setting in the total range of content. Use a slider when you want people to set defined values (such as volume or brightness), or when people would benefit from instant feedback on the effect of setting changes.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Slider(...)

Slider.shinyInput(inputId, ..., value = defaultValue)

updateSlider.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **ariaLabel** `string`
  A description of the Slider for the benefit of screen readers.
- **ariaValueText** `(value: number) => string`
  A text description of the Slider number value for the benefit of screen readers. This should be used when the Slider number value is not accurately represented by a number.
- **buttonProps** `React.HTMLAttributes<HTMLButtonElement>`
  Optional mixin for additional props on the thumb button within the slider.
- **className** `string`
  Optional className to attach to the slider root element.
- **componentRef** `IRefObject<ISlider>`
  Optional callback to access the ISlider interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultValue** `number`
  The initial value of the Slider. Use this if you intend for the Slider to be an uncontrolled component. This value is mutually exclusive to value. Use one or the other.
- **disabled** `boolean`
  Optional flag to render the Slider as disabled.
- **label** `string`
  Description label of the Slider
- **max** `number`
  The max value of the Slider
- **min** `number`
  The min value of the Slider
- **onChange** `(value: number) => void`
  Callback when the value has been changed
- **onChanged** `(event: MouseEvent | TouchEvent | KeyboardEvent, value: number) => void`
  Callback on mouse up or touch end
- **originFromZero** `boolean`
  Optional flag to attach the origin of slider to zero. Helpful when the range include negatives.
- **showValue** `boolean`
  Whether to show the value on the right of the Slider.
- **snapToStep** `boolean`
  Optional flag to decide that thumb will snap to closest value while moving the slider

- **step** `number`
  The difference between the two adjacent values of the Slider
- **styles** `IStyleFunctionOrObject<ISliderStyleProps, ISliderStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by High-Order Component.
- **value** `number`
  The initial value of the Slider. Use this if you intend to pass in a new value as a result of onChange events. This value is mutually exclusive to defaultValue. Use one or the other.
- **valueFormat** `(value: number) => string`
  Optional function to format the slider value.
- **vertical** `boolean`
  Optional flag to render the slider vertically. Defaults to rendering horizontal.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Don't use a slider for binary settings.
- Don't use a continuous slider if the range of values is large.
- Don't use for a range with fewer than three values.
- Sliders are typically horizontal but can be vertical, when needed.

Content:

- Include a label indicating what value the slider changes.
- Use step points if you don't want the slider to allow arbitrary values between minimum and maximum.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    Slider.shinyInput(ns("slider"), value = 0, min = -100, max = 100),
    textOutput(ns("sliderValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$sliderValue <- renderText({
```

```
      sprintf("Value: %s", input$slider)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

SpinButton *SpinButton*

Description

A spin button (SpinButton) allows someone to incrementally adjust a value in small steps. It's mainly used for numeric values, but other values are supported too.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
SpinButton(...)

SpinButton.shinyInput(inputId, ..., value = defaultValue)

updateSpinButton.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.
inputId	ID of the component.
value	Starting value.
session	Object passed as the session argument to Shiny server.

Details

- **ariaDescribedBy** `string`
  ID of a label which describes the control, if not using the default label.
- **ariaLabel** `string`
  A description of the control for the benefit of screen reader users.

- **ariaPositionInSet** `number`
  The position in the parent set (if in a set).
- **ariaSetSize** `number`
  The total size of the parent set (if in a set).
- **ariaValueNow** `number`
  Sets the control's aria-valuenow. This is the numeric form of `value`. Providing this only makes sense when using as a controlled component.
- **ariaValueText** `string`
- **className** `string`
  Custom className for the control.
- **componentRef** `IRefObject<ISpinButton>`
  Gets the component ref.
- **decrementButtonAriaLabel** `string`
  Accessible label text for the decrement button (for screen reader users).
- **decrementButtonIcon** `IIconProps`
  Custom props for the decrement button.
- **defaultValue** `string`
  Initial value of the control. Updates to this prop will not be respected.

Use this if you intend for the SpinButton to be an uncontrolled component which maintains its own value. Mutually exclusive with `value`.

- **disabled** `boolean`
  Whether or not the control is disabled.
- **downArrowButtonStyles** `Partial<IButtonStyles>`
  Custom styles for the down arrow button.

Note: The buttons are in a checked state when arrow keys are used to incremenent/decrement the SpinButton. Use `rootChecked` instead of `rootPressed` for styling when that is the case.

- **getClassNames** `(theme: ITheme, disabled: boolean, isFocused: boolean, keyboardSpinDirection: Keybo`
  Custom function for providing the classNames for the control. Can be used to provide all styles for the component instead of applying them on top of the default styles.
- **iconButtonProps** `IButtonProps`
  Additional props for the up and down arrow buttons.
- **iconProps** `IIconProps`
  Props for an icon to display alongside the control's label.
- **incrementButtonAriaLabel** `string`
  Accessible label text for the increment button (for screen reader users).
- **incrementButtonIcon** `IIconProps`
  Custom props for the increment button.
- **inputProps** `React.InputHTMLAttributes<HTMLElement | HTMLInputElement>`
  Additional props for the input field.
- **keytipProps** `IKeytipProps`
  Keytip for the control.

- **label** `string`
  Descriptive label for the control.
- **labelPosition** `Position`
  Where to position the control's label.
- **max** `number`
  Max value of the control.
- **min** `number`
  Min value of the control.
- **onBlur** `React.FocusEventHandler<HTMLInputElement>`
  Callback for when the control loses focus.
- **onDecrement** `(value: string, event?: React.MouseEvent<HTMLElement> | React.KeyboardEvent<HTMLElem`
  Callback for when the decrement button or down arrow key is pressed.
- **onFocus** `React.FocusEventHandler<HTMLInputElement>`
  Callback for when the user focuses the control.
- **onIncrement** `(value: string, event?: React.MouseEvent<HTMLElement> | React.KeyboardEvent<HTMLEleme`
  Callback for when the increment button or up arrow key is pressed.
- **onValidate** `(value: string, event?: React.SyntheticEvent<HTMLElement>) => string | void`
  Callback for when the entered value should be validated.
- **precision** `number`
  How many decimal places the value should be rounded to.

The default is calculated based on the precision of `step`: i.e. if step = 1, precision = 0. step = 0.0089, precision = 4. step = 300, precision = 2. step = 23.00, precision = 2.

- **step** `number`
  Difference between two adjacent values of the control. This value is used to calculate the precision of the input if no `precision` is given. The precision calculated this way will always be \>= 0.
- **styles** `Partial<ISpinButtonStyles>`
  Custom styling for individual elements within the control.
- **theme** `ITheme`
  Theme provided by HOC.
- **title** `string`
  A more descriptive title for the control, visible on its tooltip.
- **upArrowButtonStyles** `Partial<IButtonStyles>`
  Custom styles for the up arrow button.

Note: The buttons are in a checked state when arrow keys are used to incremenent/decrement the SpinButton. Use `rootChecked` instead of `rootPressed` for styling when that is the case.

- **value** `string`
  Current value of the control.

Use this if you intend to pass in a new value as a result of change events. Mutually exclusive with `defaultValue`.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a spin button when you need to incrementally change a value.
- Use a spin button when values are tied to a unit of measure.
- Don't use a spin button for binary settings.
- Don't use a spin button for a range of three values or less.
- Place labels to the left of the spin button control. For example, "Length of ruler (cm)".
- Spin button width should adjust to fit the number values.

Content:

- Include a label indicating what value the spin button changes.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    SpinButton.shinyInput(ns("spin"), value = 15, min = 0, max = 50, step = 5),
    textOutput(ns("spinValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$spinValue <- renderText({
      sprintf("Value: %s", input$spin)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Spinner` *Spinner*

Description

A Spinner is an outline of a circle which animates around itself indicating to the user that things are processing. A Spinner is shown when it's unsure how long a task will take making it the indeterminate version of a ProgressIndicator. They can be various sizes, located inline with content or centered. They generally appear after an action is being processed or committed. They are subtle and generally do not take up much space, but are transitions from the completed task.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Spinner(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaLabel** `string`
  Alternative status label for screen reader
- **ariaLive** `'assertive' | 'polite' | 'off'`
  Politeness setting for label update announcement.
- **className** `string`
  Additional CSS class(es) to apply to the Spinner.
- **componentRef** `IRefObject<ISpinner>`
  Optional callback to access the ISpinner interface. Use this instead of ref for accessing the public methods and properties of the component.
- **label** `string`
  The label to show next to the Spinner. Label updates will be announced to the screen readers. Use ariaLive to control politeness level.
- **labelPosition** `SpinnerLabelPosition`
  The position of the label in regards of the spinner animation.
- **size** `SpinnerSize`
  The size of Spinner to render. { extraSmall, small, medium, large }
- **styles** `IStyleFunctionOrObject<ISpinnerStyleProps, ISpinnerStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme (provided through customization.)

- **type** SpinnerType
  Deprecated and will be removed at \>= 2.0.0. Use SpinnerSize instead.

Value

Object with shiny.tag class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Spinner(size = 3, label = "Loading, please wait...")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

Stack *Stack*

Description

A Stack is a container-type component that abstracts the implementation of a flexbox in order to define the layout of its children components.

Stack Properties:

Although the Stack component has a number of different properties, there are three in particular that define the overall layout that the component has:

1. Direction: Refers to whether the stacking of children components is horizontal or vertical. By default the Stack component is vertical, but can be turned horizontal by adding the horizontal property when using the component.
2. Alignment: Refers to how the children components are aligned inside the container. This is controlled via the verticalAlign and horizontalAlign properties. One thing to notice here is that while flexbox containers align always across the cross axis, Stack aims to remove the mental strain involved in this process by making the verticalAlign and horizontalAlign properties always follow the vertical and horizontal axes, respectively, regardless of the direction of the Stack.
3. Spacing: Refers to the space that exists between children components inside the Stack. This is controlled via the gap and verticalGap properties.

Stack Items:

The `Stack` component provides an abstraction of a flexbox container but there are some flexbox related properties that are applied on specific children of the flexbox instead of being applied on the container. This is where `Stack Items` comes into play.

A `Stack Item` abstracts those properties that are or can be specifically applied on flexbox's children, like `grow` and `shrink`.

To use a `Stack Item` in an application, the `Stack` component should be imported and `Stack.Item` should be used inside of a `Stack`. This is done so that the existence of the `Stack Item` is inherently linked to the `Stack` component.

Stack Wrapping:

Aside from the previously mentioned properties, there is another property called `wrap` that determines if items overflow the `Stack` container or wrap around it. The wrap property only works in the direction of the `Stack`, which means that the children components can still overflow in the perpendicular direction (i.e. in a `Vertical Stack`, items might overflow horizontally and vice versa).

Stack Nesting:

`Stacks` can be nested inside one another in order to be able to configure the layout of the application as desired.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Stack(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **align** `'auto' | 'stretch' | 'baseline' | 'start' | 'center' | 'end'`
  Defines how to align the StackItem along the x-axis (for vertical Stacks) or the y-axis (for horizontal Stacks).
- **className** `string`
  Defines a CSS class name used to style the StackItem.
- **disableShrink** `boolean`
  Defines whether the StackItem should be prevented from shrinking. This can be used to prevent a StackItem from shrinking when it is inside of a Stack that has shrinking items.
- **grow** `boolean | number | 'inherit' | 'initial' | 'unset'`
  Defines how much to grow the StackItem in proportion to its siblings.
- **order** `number | string`
  Defines order of the StackItem.
- **shrink** `boolean | number | 'inherit' | 'initial' | 'unset'`
  Defines at what ratio should the StackItem shrink to fit the available space.

- **verticalFill** `boolean`
  Defines whether the StackItem should take up 100% of the height of its parent.
- **as** `React.ElementType<React.HTMLAttributes<HTMLElement>>`
  Defines how to render the Stack.
- **disableShrink** `boolean`
  Defines whether Stack children should not shrink to fit the available space.
- **gap** `number | string`
  Defines the spacing between Stack children. The property is specified as a value for 'row gap', followed optionally by a value for 'column gap'. If 'column gap' is omitted, it's set to the same value as 'row gap'.
- **grow** `boolean | number | 'inherit' | 'initial' | 'unset'`
  Defines how much to grow the Stack in proportion to its siblings.
- **horizontal** `boolean`
  Defines whether to render Stack children horizontally.
- **horizontalAlign** `Alignment`
  Defines how to align Stack children horizontally (along the x-axis).
- **maxHeight** `number | string`
  Defines the maximum height that the Stack can take.
- **maxWidth** `number | string`
  Defines the maximum width that the Stack can take.
- **padding** `number | string`
  Defines the inner padding of the Stack.
- **reversed** `boolean`
  Defines whether to render Stack children in the opposite direction (bottom-to-top if it's a vertical Stack and right-to-left if it's a horizontal Stack).
- **verticalAlign** `Alignment`
  Defines how to align Stack children vertically (along the y-axis).
- **verticalFill** `boolean`
  Defines whether the Stack should take up 100% of the height of its parent. This property is required to be set to true when using the `grow` flag on children in vertical oriented Stacks. Stacks are rendered as block elements and grow horizontally to the container already.
- **wrap** `boolean`
  Defines whether Stack children should wrap onto multiple rows or columns when they are about to overflow the size of the Stack.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
```

```
  ns <- NS(id)
  Stack(
    tokens = list(childrenGap = 10),
    reversed = TRUE,
    span("Item One"),
    span("Item Two"),
    span("Item Three")
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`SwatchColorPicker` *SwatchColorPicker*

Description

A swatch color picker (`SwatchColorPicker`) displays color options as a grid. It can be shown by itself, with a header and dividers, or as a button that expands to show the swatch color picker.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
SwatchColorPicker(...)

SwatchColorPicker.shinyInput(inputId, ..., value = defaultValue)

updateSwatchColorPicker.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **color** `string`
  The CSS-compatible string to describe the color
- **id** `string`
  Arbitrary unique string associated with this option
- **index** `number`
  Index for this option
- **label** `string`
  Tooltip and aria label for this item
- **borderWidth** `number`
  Width of the border that indicates a selected/hovered cell, in pixels.
- **circle** `boolean`
  True if this cell should be rendered as a circle, false if it should be a square. @default `true` (render as circle)
- **color** `string`
  The CSS-compatible string to describe the color
- **disabled** `boolean`
  Whether this cell should be disabled @default false
- **height** `number`
  Height of the cell, in pixels
- **id** `string`
  Used as a PREFIX for the cell's ID (the cell will not have this literal string as its ID).
- **idPrefix** `string`
  Prefix for this cell's ID. Will be required in a future version once `id` is removed.
- **index** `number`
  Index for this option
- **item** `IColorCellProps`
  Item to render
- **label** `string`
  Tooltip and aria label for this item
- **onClick** `(item: IColorCellProps) => void`
  Handler for when a color cell is clicked.
- **onFocus** `(item: IColorCellProps) => void`
- **onHover** `(item?: IColorCellProps) => void`
- **onKeyDown** `(ev: React.KeyboardEvent<HTMLButtonElement>) => void`
- **onMouseEnter** `(ev: React.MouseEvent<HTMLButtonElement>) => boolean`
  Mouse enter handler. Returns true if the event should be processed, false otherwise.
- **onMouseLeave** `(ev: React.MouseEvent<HTMLButtonElement>) => void`

- **onMouseMove** `(ev: React.MouseEvent<HTMLButtonElement>) => boolean`
  Mouse move handler. Returns true if the event should be processed, false otherwise.
- **onWheel** `(ev: React.MouseEvent<HTMLButtonElement>) => void`
- **selected** `boolean`
  Whether this cell is currently selected
- **styles** `IStyleFunctionOrObject<IColorPickerGridCellStyleProps, IColorPickerGridCellStyles>`
  Custom styles for the component.
- **theme** `ITheme`
  The theme object to use for styling.
- **width** `number`
  Width of the cell, in pixels
- **ariaPosInSet** `number`
  Position this grid is in the parent set (index in a parent menu, for example)
- **ariaSetSize** `number`
  Size of the parent set (size of parent menu, for example)
- **cellBorderWidth** `number`
  Width of the border indicating a hovered/selected cell, in pixels
- **cellHeight** `number`
  Height of an individual cell, in pixels
- **cellMargin** `number`
  The distance between cells, in pixels
- **cellShape** `'circle' | 'square'`
  The shape of the color cells. @default 'circle'
- **cellWidth** `number`
  Width of an individual cell, in pixels
- **className** `string`
  Additional class name to provide on the root element
- **colorCells** `IColorCellProps[]`
  The color cells that will be made available to the user.

Note: When the reference to this prop changes, regardless of how many color cells change, all of the color cells will be re-rendered (potentially bad perf) because we memoize based on this prop's reference.

- **columnCount** `number`
  Number of columns for the swatch color picker
- **disabled** `boolean`
  Whether the control is disabled.
- **doNotContainWithinFocusZone** `boolean`
  If false (the default), the grid is contained inside a FocusZone. If true, a FocusZone is not used. @default false
- **focusOnHover** `boolean`
  Whether to update focus when a cell is hovered.

- **getColorGridCellStyles** `IStyleFunctionOrObject<IColorPickerGridCellStyleProps, IColorPickerGridCel`
  Styles for the grid cells.
- **id** `string`
  ID for the swatch color picker's root element. Also used as a prefix for the IDs of color cells.
- **isControlled** `boolean`
  Indicates whether the SwatchColorPicker is fully controlled. When true, the component will not set its internal state to track the selected color. Instead, the parent component will be responsible for handling state in the callbacks like `onColorChanged`.

NOTE: This property is a temporary workaround to force the component to be fully controllable without breaking existing behavior

- **mouseLeaveParentSelector** `string | undefined`
  Selector to focus on mouse leave. Should only be used in conjunction with `focusOnHover`.
- **onCellFocused** `(id?: string, color?: string) => void`
  Callback for when the user focuses a color cell. If `id` and `color` are unspecified, cells are no longer being focused.
- **onCellHovered** `(id?: string, color?: string) => void`
  Callback for when the user hovers over a color cell. If `id` and `color` are unspecified, cells are no longer being hovered.
- **onColorChanged** `(id?: string, color?: string) => void`
  Callback for when the user changes the color. If `id` and `color` are unspecified, there is no selected cell. (e.g. the user executed the currently selected cell to unselect it)
- **positionInSet** `number`
- **selectedId** `string`
  The ID of color cell that is currently selected
- **setSize** `number`
- **shouldFocusCircularNavigate** `boolean`
  Whether focus should cycle back to the beginning once the user navigates past the end (and vice versa). Only relevant if `doNotContainWithinFocusZone` is not true.
- **styles** `IStyleFunctionOrObject<ISwatchColorPickerStyleProps, ISwatchColorPickerStyles>`
  Styles for the component.
- **theme** `ITheme`
  Theme to apply to the component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- Use a swatch color picker when there are multiple color options that can be grouped or collapsed under one title.
- Don't use a swatch color picker when there's a large number of color options. The best component for that is a color picker.

Examples

```
library(shiny)
library(shiny.fluent)

colorCells <- list(
  list(id = "orange", color = "#ca5010"),
  list(id = "cyan", color = "#038387"),
  list(id = "blueMagenta", color = "#8764b8"),
  list(id = "magenta", color = "#881798"),
  list(id = "white", color = "#ffffff")
)

ui <- function(id) {
  ns <- NS(id)
  div(
    SwatchColorPicker.shinyInput(ns("color"), value = "orange",
      colorCells = colorCells, columnCount = length(colorCells)
    ),
    textOutput(ns("swatchValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$swatchValue <- renderText({
      sprintf("Value: %s", input$color)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`TeachingBubble` *TeachingBubble*

Description

A teaching bubble (`TeachingBubble`) brings attention to a new or important feature, with the goal of increasing engagement with the feature. A teaching bubble typically follows a coach mark.

Use teaching bubbles sparingly. Consider how frequently people will see it, and how many they'll see across their entire experience.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
TeachingBubble(...)

TeachingBubbleContent(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **ariaDescribedBy** `string`
  Defines the element id referencing the element containing the description for the TeachingBubble.
- **ariaLabelledBy** `string`
  Defines the element id referencing the element containing label text for TeachingBubble.
- **calloutProps** `ICalloutProps`
  Properties to pass through for Callout, reference detail properties in ICalloutProps
- **componentRef** `IRefObject<ITeachingBubble>`
  Optional callback to access the ITeachingBubble interface. Use this instead of ref for accessing the public methods and properties of the component.
- **focusTrapZoneProps** `IFocusTrapZoneProps`
  Properties to pass through for FocusTrapZone, reference detail properties in IFocusTrapZoneProps
- **footerContent** `string | JSX.Element`
  Text that will be rendered in the footer of the TeachingBubble. May be rendered alongside primary and secondary buttons.
- **hasCloseButton** `boolean`
  Whether the TeachingBubble renders close button in the top right corner.
- **hasCloseIcon** `boolean`
- **hasCondensedHeadline** `boolean`
  A variation with smaller bold headline and no margins.
- **hasSmallHeadline** `boolean`
  A variation with smaller bold headline and margins to the body. (`hasCondensedHeadline` takes precedence if it is also set to true.)
- **headline** `string`
  A headline for the Teaching Bubble.
- **illustrationImage** `IImageProps`
  An Image for the TeachingBubble.

- **isWide** `boolean`
  Whether or not the TeachingBubble is wide, with image on the left side.
- **onDismiss** `(ev?: any) => void`
  Callback when the TeachingBubble tries to close.
- **primaryButtonProps** `IButtonProps`
  The Primary interaction button
- **secondaryButtonProps** `IButtonProps`
  The Secondary interaction button
- **styles** `IStyleFunctionOrObject<ITeachingBubbleStyleProps, ITeachingBubbleStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **target** `Target`
  Element, MouseEvent, Point, or querySelector string that the TeachingBubble should anchor to.
- **targetElement** `HTMLElement`
- **theme** `ITheme`
  Theme provided by High-Order Component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Layout:

- Teaching bubbles can be used in sequence to walk people through complex, multistep interactions only. And only show one teaching bubble at a time.
- A maximum of no more than 3 sequenced teaching bubbles should be used in a single experience.
- Sequenced teaching bubbles should have "x of y" text to indicate progress through the sequence. For example, the first teaching bubble in a sequence might be "1 of 3".)
- Always place the primary button on the left, the secondary button just to the right of it.
- Show only one primary button that inherits theme color at rest state. In the event there are more than two buttons with equal priority, all buttons should have neutral backgrounds.

Content:

A teaching bubble should include:

Title:
You have 25 characters (including spaces) to draw people in and get them interested. Limit to one line of text, and use sentence casing (capitalize only the first word and any proper nouns) with no punctuation.

Body copy:
Lead with why the feature is useful rather than describe what it is. What does it make possible? Keep it within 120 characters (including spaces).

Action buttons:
Limit button labels to 15 characters (including spaces). Provide people with an explicit action to dismiss the teaching bubble, such as “Got it”. Include a secondary button to give people the option to take action, such as “Show me” or “Edit settings”. When two buttons are needed, make the call-to-action button the primary button and the dismissal button (such as “No thanks”) the secondary button. Use sentence casing (capitalize only the first word and any proper nouns) with no punctuation. On a sequenced teaching bubble, use "Next" for the action button label and "Got it" for the final sequenced teaching bubble action button with text that closes the experience.

Link (Optional):
If there are additional steps people need to know about, or helpful information they may want to read, consider linking to a help article. Typically, these links are labeled “Learn more” with no punctuation at the end.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    DefaultButton.shinyInput(
      ns("toggleTeachingBubble"),
      id = "target",
      text = "Toggle TeachingBubble"
    ),
    reactOutput(ns("teachingBubble"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    showBubble <- reactiveVal(FALSE)
    observeEvent(input$toggleTeachingBubble, showBubble(!showBubble()))
    output$teachingBubble <- renderReact({
      if (showBubble()) {
        TeachingBubble(
          target = "#target",
          headline = "Very useful!"
        )
      }
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Text` *Text*

Description

Text is a component for displaying text. You can use Text to standardize text across your web app.

You can specify the `variant` prop to apply font styles to Text. This variant pulls from the Fluent UI React theme loaded on the page. If you do not specify the `variant` prop, by default, Text applies the styling from specifying the `variant` value to `medium`.

The Text control is inline wrap by default. You can specify `block` to enable block and `nowrap` to enable `nowrap`. For ellipsis on overflow to work properly, `block` and `nowrap` should be manually set to `true`.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Text(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **as** `React.ElementType<React.HTMLAttributes<HTMLElement>>`
  Optionally render the component as another component type or primitive.
- **block** `boolean`
  Whether the text is displayed as a block element.

Note that in order for ellipsis on overflow to work properly, `block` and `nowrap` should be set to true.

- **nowrap** `boolean`
  Whether the text is not wrapped.

Note that in order for ellipsis on overflow to work properly, `block` and `nowrap` should be set to true.

- **variant** `keyof IFontStyles`
  Optional font type for Text.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  Text(variant = "xLarge", "Some text with a nice Fluent UI font")
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`ThemeProvider` *Theme*

Description

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

`ThemeProvider` is a utility that applies contextual theming to its child components. See the official docs for details.

Usage

```
ThemeProvider(...)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Examples

```
# Example 1
library(shiny)
library(shiny.fluent)

options <- list(
  list(key = "A", text = "Option A"),
```

```
  list(key = "B", text = "Option B")
)
theme <- list(
  palette = list(
    themePrimary = "#8dd400",
    themeLighterAlt = "#060800",
    themeLighter = "#172200",
    themeLight = "#2a3f00",
    themeTertiary = "#557f00",
    themeSecondary = "#7cba00",
    themeDarkAlt = "#97d816",
    themeDark = "#a6de35",
    themeDarker = "#bce766",
    neutralLighterAlt = "#323130",
    neutralLighter = "#31302f",
    neutralLight = "#2f2e2d",
    neutralQuaternaryAlt = "#2c2b2a",
    neutralQuaternary = "#2a2928",
    neutralTertiaryAlt = "#282726",
    neutralTertiary = "#c8c8c8",
    neutralSecondary = "#d0d0d0",
    neutralPrimaryAlt = "#dadada",
    neutralPrimary = "#ffffff",
    neutralDark = "#f4f4f4",
    black = "#f8f8f8",
    white = "#323130"
  )
)

ui <- function(id) {
  ns <- NS(id)
  ThemeProvider(
    theme = theme,
    Stack(
      tokens = list(childrenGap = "10px"),
      style = list(width = 250),
      PrimaryButton(text = "PrimaryButton"),
      Checkbox(label = "Checkbox"),
      ChoiceGroup(label = "ChoiceGroup", options = options)
    )
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}

# Example 2
library(shiny)
```

```
library(shiny.fluent)

options <- list(
  list(key = "A", text = "Option A"),
  list(key = "B", text = "Option B")
)

# Use JSON created in Theme Designer
# https://fabricweb.z5.web.core.windows.net/pr-deploy-site/refs/heads/master/theming-designer/
theme <- '{
  "themePrimary": "#324f09",
  "themeLighterAlt": "#dfead1",
  "themeLighter": "#c4d7ab",
  "themeLight": "#abc388",
  "themeTertiary": "#92b069",
  "themeSecondary": "#7c9c4e",
  "themeDarkAlt": "#678937",
  "themeDark": "#547624",
  "themeDarker": "#426214",
  "neutralLighterAlt": "#f8ebce",
  "neutralLighter": "#f4e8cb",
  "neutralLight": "#eadec2",
  "neutralQuaternaryAlt": "#dacfb5",
  "neutralQuaternary": "#d0c5ad",
  "neutralTertiaryAlt": "#c8bea6",
  "neutralTertiary": "#595858",
  "neutralSecondary": "#373636",
  "neutralPrimaryAlt": "#2f2e2e",
  "neutralPrimary": "#000000",
  "neutralDark": "#151515",
  "black": "#0b0b0b",
  "white": "#fff2d4"
}'

ui <- function(id) {
  ns <- NS(id)
  ThemeProvider(
    theme = parseTheme(json = theme),
    Stack(
      tokens = list(childrenGap = "10px"),
      style = list(width = 250),
      PrimaryButton(text = "PrimaryButton"),
      Checkbox(label = "Checkbox"),
      ChoiceGroup(label = "ChoiceGroup", options = options)
    )
  )
}

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

if (interactive()) {
```

```
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`Toggle` *Toggle*

Description

A toggle represents a physical switch that allows someone to choose between two mutually exclusive options. For example, "On/Off", "Show/Hide". Choosing an option should produce an immediate result.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
Toggle(...)

Toggle.shinyInput(inputId, ..., value = defaultValue)

updateToggle.shinyInput(
  session = shiny::getDefaultReactiveDomain(),
  inputId,
  ...
)
```

Arguments

`...`	Props to pass to the component. The allowed props are listed below in the **Details** section.
`inputId`	ID of the component.
`value`	Starting value.
`session`	Object passed as the `session` argument to Shiny server.

Details

- **ariaLabel** `string`
  Text for screen-reader to announce as the name of the toggle.
- **as** `IComponentAs<React.HTMLAttributes<HTMLElement>>`
  Render the root element as another type.
- **checked** `boolean`
  Checked state of the toggle. If you are maintaining state yourself, use this property. Otherwise use `defaultChecked`.

- **componentRef** `IRefObject<IToggle>`
  Optional callback to access the IToggle interface. Use this instead of ref for accessing the public methods and properties of the component.
- **defaultChecked** `boolean`
  Initial state of the toggle. If you want the toggle to maintain its own state, use this. Otherwise use `checked`.
- **disabled** `boolean`
  Optional disabled flag.
- **inlineLabel** `boolean`
  Whether the label (not the onText/offText) should be positioned inline with the toggle control. Left (right in RTL) side when on/off text provided VS right (left in RTL) side when no on/off text. Caution: when not providing on/off text user may get confused in differentiating the on/off states of the toggle.
- **keytipProps** `IKeytipProps`
  Optional keytip for this toggle
- **label** `string | JSX.Element`
  A label for the toggle.
- **offAriaLabel** `string`
- **offText** `string`
  Text to display when toggle is OFF. Caution: when not providing on/off text user may get confused in differentiating the on/off states of the toggle.
- **onAriaLabel** `string`
- **onChange** `(event: React.MouseEvent<HTMLElement>, checked?: boolean) => void`
  Callback issued when the value changes.
- **onChanged** `(checked: boolean) => void`
- **onText** `string`
  Text to display when toggle is ON. Caution: when not providing on/off text user may get confused in differentiating the on/off states of the toggle.
- **role** `'checkbox' | 'switch' | 'menuitemcheckbox'`
  (Optional) Specify whether to use the "switch" role (ARIA 1.1) or the checkbox role (ARIA 1.0). If unspecified, defaults to "switch".
- **styles** `IStyleFunctionOrObject<IToggleStyleProps, IToggleStyles>`
  Optional styles for the component.
- **theme** `ITheme`
  Theme provided by HOC.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app. The update functions return nothing (called for side effects).

Best practices

Layout:

- When people need to perform extra steps for changes to take effect, use a check box instead. For example, if they must click a "Submit", "Next", or "OK" button to apply changes, use a check box.

Content:

- Only replace the On/Off labels if there are more specific labels for the setting. For example, you might use Show/Hide if the setting is "Show images".
- Keep descriptive text short and concise—two to four words; preferably nouns. For example, "Focused inbox" or "WiFi".

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  div(
    Toggle.shinyInput(ns("toggle"), value = TRUE),
    textOutput(ns("toggleValue"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    output$toggleValue <- renderText({
      sprintf("Value: %s", input$toggle)
    })
  })
}

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

TooltipHost *Tooltip*

Description

A good tooltip briefly describes unlabeled controls or provides a bit of additional information about labeled controls, when this is useful. It can also help customers navigate the UI by offering additional—not redundant—information about control labels, icons, and links. A tooltip should always add valuable information; use sparingly.

There are two components you can use to display a tooltip:

- **Tooltip:** A styled tooltip that you can display on a chosen target.
- **TooltipHost:** A wrapper that automatically shows a tooltip when the wrapped element is hovered or focused.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
TooltipHost(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Details

- **calloutProps** `ICalloutProps`
  Additional properties to pass through for Callout.
- **className** `string`
  Class name to apply to the *tooltip itself*, not the host. To apply a class to the host, use `hostClassName` or `styles.root`.
- **closeDelay** `number`
  Number of milliseconds to delay closing the tooltip, so that the user has time to hover over the tooltip and interact with it. Hovering over the tooltip will count as hovering over the host, so that the tooltip will stay open if the user is actively interacting with it.
- **componentRef** `IRefObject<ITooltipHost>`
  Optional callback to access the ITooltipHost interface. Use this instead of ref for accessing the public methods and properties of the component.
- **content** `string | JSX.Element | JSX.Element[]`
  Content to display in the Tooltip.
- **delay** `TooltipDelay`
  Length of delay before showing the tooltip on hover.
- **directionalHint** `DirectionalHint`
  How the tooltip should be anchored to its `targetElement`.
- **directionalHintForRTL** `DirectionalHint`
  How the element should be positioned in RTL layouts. If not specified, a mirror of `directionalHint` will be used.
- **hostClassName** `string`
  Class name to apply to tooltip host.
- **id** `string`
  Optional ID to pass through to the tooltip (not used on the host itself). Auto-generated if not provided.
- **onTooltipToggle** `onTooltipToggle?(isTooltipVisible: boolean): void;`
  Notifies when tooltip becomes visible or hidden, whatever the trigger was.

- **overflowMode** `TooltipOverflowMode`
  If this is unset (the default), the tooltip is always shown even if there's no overflow.

If set, only show the tooltip if the specified element (`Self` or `Parent`) has overflow. When set to `Parent`, the parent element is also used as the tooltip's target element.

Note that even with `Self` mode, the TooltipHost *does not* check whether any children have overflow.

- **setAriaDescribedBy** `boolean`
  Whether or not to mark the TooltipHost root element as described by the tooltip. If not specified, the caller should pass an `id` to the TooltipHost (to be passed through to the Tooltip) and mark the appropriate element as `aria-describedby` the `id`.
- **styles** `IStyleFunctionOrObject<ITooltipHostStyleProps, ITooltipHostStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **theme** `ITheme`
  Theme provided by higher-order component.
- **tooltipProps** `ITooltipProps`
  Additional properties to pass through for Tooltip.
- **calloutProps** `ICalloutProps`
  Properties to pass through for Callout.
- **componentRef** `IRefObject<ITooltip>`
  Optional callback to access the ITooltip interface. Use this instead of ref for accessing the public methods and properties of the component.
- **content** `string | JSX.Element | JSX.Element[]`
  Content to be passed to the tooltip
- **delay** `TooltipDelay`
  Length of delay. Set to `TooltipDelay.zero` if you do not want a delay.
- **directionalHint** `DirectionalHint`
  How the tooltip should be anchored to its `targetElement`.
- **directionalHintForRTL** `DirectionalHint`
  How the element should be positioned in RTL layouts. If not specified, a mirror of `directionalHint` will be used instead
- **maxWidth** `string | null`
  Max width of tooltip
- **onRenderContent** `IRenderFunction<ITooltipProps>`
  Render function to populate tooltip content.
- **styles** `IStyleFunctionOrObject<ITooltipStyleProps, ITooltipStyles>`
  Call to provide customized styling that will layer on top of the variant rules.
- **targetElement** `HTMLElement`
  Element to anchor the Tooltip to.
- **theme** `ITheme`
  Theme provided by higher-order component.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Content:

- Don't use a tooltip to restate a button name that's already shown in the UI.
- When a control or UI element is unlabeled, use a simple, descriptive noun phrase. For example: "Highlighting pen". Only capitalize the first word (unless a subsequent word is a proper noun), and don't use a period.
- For a disabled control that could use an explanation, provide a brief description of the state in which the control will be enabled. For example: "This feature is available for line charts."
- Only use periods for complete sentences.

For a UI label that needs some explanation:

- Briefly describe what you can do with the UI element.
- Use the imperative verb form. For example, "Find text in this file" (not "Finds text in this file").
- Don't include end punctuation unless there is at least one complete sentence.

For a truncated label or a label that's likely to truncate in some languages:

- Provide the untruncated label in the tooltip.
- Don't provide a tooltip if the untruncated info is provided elsewhere on the page or flow.
- Optional: On another line, provide a clarifying description, but only if needed.

Examples

```
library(shiny)
library(shiny.fluent)

ui <- function(id) {
  ns <- NS(id)
  TooltipHost(
    content = "This is the tooltip content",
    delay = 0,
    Text("Hover over me")
  )
}

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

if (interactive()) {
  shinyApp(ui("app"), function(input, output) server("app"))
}
```

`VerticalDivider` *Divider*

Description

A Divider is a line that is used to visually differentiate different parts of a UI. They are commonly used in headers and command bars. This divider automatically center algins itself within the parent container and can be customized to be shown in different heights and colors.

For more details and examples visit the official docs. The R package cannot handle each and every case, so for advanced use cases you need to work using the original docs to achieve the desired result.

Usage

```
VerticalDivider(...)
```

Arguments

...	Props to pass to the component. The allowed props are listed below in the **Details** section.

Value

Object with `shiny.tag` class suitable for use in the UI of a Shiny app.

Best practices

Use a divider component to show a sectional or continuity change in the content between two blocks of information. The spacing around the divider is generally determined by the content surrounding it.

There are two recommended divider color combinations:

1. #C8C8C8/neutralTertiaryAlt divider when used within an #F4F4F4/neutralLighter layout
2. #EAEAEA/neutralLight divider when used within an #FFFFFF/white layout

Index