Package: rsx (via r-universe)

October 22, 2024

Title Create Encapsulated Shiny Components

Version 0.0.0.9000

Description Create complex yet manageable Shiny applications with nestable components that encapsulate both UI and state. Heavily inspired by many JavaScript frameworks, particularly Vue.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.1.9000

Imports cli, htmltools, rlang, sass, shiny

Suggests covr, knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://elianhugh.github.io/rsx/

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

RemoteUrl https://github.com/ElianHugh/rsx

RemoteRef HEAD

RemoteSha 50173cdc4a1b053f098f8f6c34ef3d7d1980ccdd

Contents

`component` *Create a component*

Description

Components represent the encapsulation of a shiny module's logic and UI into a singular object, and can be used like any other shiny tag.

For more information on the data, methods, template, and styles arguments, see related documentation.

Usage

```
component(
  name = NULL,
  data = NULL,
  methods = NULL,
  template = NULL,
  styles = NULL
)
```

Arguments

`name`	component name
`data`	a function that returns a named list of values, which are used to store the component's state
`methods`	named list of functions, which define the behavior of the component
`template`	function that returns a taglist
`styles`	function that returns a character vector or list of CSS styles that are scoped to the component

See Also

Other components: `component-data`, `component-methods`, `component-styles`, `component-template`

Examples

```
library(shiny)
counter <- component(
    name = "counter",
    data = function() {
        list(
```

```
            label = "Counter",
            count = reactiveVal(0L)
        )
    },
    template = function(ns) {
        tagList(
            actionButton(ns("button"), label = self$label),
            verbatimTextOutput(ns("out"))
        )
    },
    methods = list(
        setup = function(input, output, session) {
            observeEvent(input$button, {
                self$count(
                    self$count() + 1L
                )
            })
            output$out <- renderText(
                self$count()
            )
        }
    )
)
tagList(
 tags$h1("Counter"),
 counter()
)
```

---

component-data *Component Data*

Description

Data is used for a component's internal state and can also be used to pass information from a parent component to its children. Both component templates and methods have access to component data.

Details

Creating data:

To create component data, define a function that returns a list of named objects. These objects can be Shiny reactive objects, data frames, lists, or any other R object.

For instance, the following is a valid data function:

```
function() {
 list(
   rctv = shiny::reactiveVal(),
   df   = mtcars
 )
}
```

Accessing data:

Data can be accessed in both the methods and template parts of the component via the `self` keyword. For example, the following component defines the data `foo`, and accesses it in the template via `self$foo`.

```
x <- component(
    name = "data_access",
    data = function() {
        list(
            foo = 1L
        )
    },
    template = function(ns) {
        self$foo
    }
)

print(x())
#> <rsx::instance> `data_access`
#>   1
```

Passing data:

To pass data to a component, use the `data` argument when calling the component function. The `data` argument should be a list of named objects that match the names of the objects defined in the component's data function. For example:

```
x <- component(
    name = "data_access",
    data = function() {
        list(
            foo = 1L
        )
    },
    template = function(ns) {
        self$foo
    }
)

print(x(data = list(foo = 2L)))
#> <rsx::instance> `data_access`
#>   2
```

See Also

Other components: `component-methods`, `component-styles`, `component-template`, `component()`

component-methods *Component Methods*

Description

Methods are a list of functions that are contained within a given component, and typically will manipulate component data or respond to user input.

Methods can be accessed in both other methods and the component template via the `self` keyword.

Details

Methods are defined in the component definition as a named list of functions.

For example, the following is a method section from a simple counter component that defines two methods, setup and increment, which respectively set up the module server and increment the count state:

```
methods = list(
    setup = function(input, output, session) {
        output$txt <- renderText({
            paste0("Count is: ", self$count())
        })
        observeEvent(input$button, {
            self$count(self$count() + 1L)
        })
    },
    increment = function() {
        self$count(self$count() + 1L)
    }
)
```

Hooks:

Setup:
The setup hook is defined by passing a function named "setup" to the methods list. Setup is used as the module server for the component. This method is called when the module is first initialized, and is used to set up input/output bindings and any other necessary initialization code.

```
setup = function(input, output, session) {

}
```

Render:
The `render` hook is defined by passing a function named "render" to the methods list. The render hook allows for the modification of template code prior to its rendering.

```
render = function(element) {

}
```

See Also

Other components: component-data, component-styles, component-template, component()

---

component-styles *Component Styles*

Description

The styles argument is function that returns a character vector or list that defines the styles for the component. Styles are scoped to the component.

Details

The styles argument takes a character vector of length 1 that defines the styles for the component. For example:

```
styles = function() {
"a { color: red; }"
}
```

This would define the CSS styles for the anchor elements (<a>) in the component, setting their color to red.

Scoped Styles

Styles defined in a component are scoped to the component, meaning they will only apply to elements within that component.

To style the top-level node of the component, we can apply the styles without specifying a tag:

```
x <- component(
 name = "scoped_styles",
 template = function(ns) {
     shiny::div(
      "Hello world!"
     )
 },
 styles = function() {
    "color: red"
  }
)
```

See Also

Other components: component-data, component-methods, component-template, component()

component-template *Component Template*

Description

The template refers to the UI-generator of a given component. This is analagous to the UI function of a given shiny module.

Details

The component template function must be of the following structure (note the ns argument):

```
function(ns) {
 # must either return an object that can
 # be coerced to a `shiny::tags` object or a `shiny::tagList`
}
```

The following is an example of a valid template:

```
function(ns) {
 shiny::div("Hello world!")
}
```

Namespacing

The template function requires one argument: `ns`. `ns` is used identically to shiny modules, and helps distinguish between component instances.

Slots

Component templates can be nested through the use of slots. Slots are placeholder elements that tell rsx where tags should be placed.

```
x <- component(
    name = "slots",
    template = function(ns) {
        shiny::tagList(
            shiny::tags$slot(),
            shiny::p("bar")
        )
    }
)

print(x(shiny::p("foo")))
#> <rsx::instance> `slots`
#>   <p>foo</p>
#>   <p>bar</p>
```

You can also specify if you'd like content to be used in the case that a slot isn't filled – this is typically called "fallback" content.

```
x <- component(
    name = "fallback",
    template = function(ns) {
        shiny::tags$slot(
            shiny::p("Hello!")
        )
    }
)

print(x())
#> <rsx::instance> `fallback`
#>   <p>Hello!</p>
```

Named Slots:

Slots can be further distinguished by name attributes, which can be used to target specific areas of the template.

```
x <- component(
    name = "named_slots",
    template = function(ns) {
        shiny::tagList(
            shiny::tags$slot(name = "a"),
            shiny::tags$slot(name = "b")
        )
    }
)
print(x(shiny::p("bar", slot = "b"), shiny::p("foo", slot = "a")))
#> <rsx::instance> `named_slots`
#>   <p>foo</p>
#>   <p>bar</p>
```

See Also

Other components: component-data, component-methods, component-styles, component()

`decompose` *Decompose a component instance*

Description

Given a component instance tag `x`, decompose the instance into separate server and UI elements.

Usage

```
decompose(x)
```

Arguments

x	a shiny tag returned from calling a component

Value

list

Examples

```
comp <- component(
   name = "decompose",
   template = function(ns) {
       shiny::div("hello world")
   },
   methods = list(
       setup = function(input, output, session) {
           # noop
       }
   )
)
x <- decompose(comp())
print(x)
```

---

is.component *Is a component*

Description

Is a component

Usage

```
is.component(x)
```

Arguments

x	object

is.instance_tag *Is an instance tag*

Description

Is an instance tag

Usage

```
is.instance_tag(x)
```

Arguments

x	object

rsx_app *Create an rsx app object*

Description

Create a new instance of an rsx application by passing a top level rsx::component as the application root. This is analagous to running `shiny::shinyApp()`.

Usage

```
rsx_app(root, ..., resource_path = NULL, app_class = "App")
```

Arguments

root	an `rsx::component` object
...	further arguments passed to `shiny::shinyApp`
resource_path	path to a resource folder, if `NULL` styles will be inlined
app_class	the html class attribute for the app wrapper

See Also

Other compilation: `rsx_server()`, `rsx_ui()`

rsx_module_server *Create an rsx module server*

Description

rsx_module_server is a low-level function that loads up the module servers of all instantiated components. This is similar to rsx_server, but also allows for namespacing the rsx server.

Usage

```
rsx_module_server(id)
```

Arguments

id	unique namespace

rsx_server *Create an rsx server*

Description

rsx_server is a low-level function that loads up the module servers of all instantiated rsx::components. If the rsx_app function is not viable for your shiny application setup, the rsx_server function can be used.

Usage

```
rsx_server()
```

Details

Some things to note:

- Due to randomised hashing of component namepaces, shiny modules are not nested in rsx
- If a component does not have a setup function, a module server will not be created for the instance

Value

shiny server function

See Also

Other compilation: rsx_app(), rsx_ui()

rsx_ui *Create an rsx UI taglist*

Description

`rsx_ui` is a low-level function for creating rsx ui without the use of `rsx_app`.

Usage

```
rsx_ui(root, id = NULL, app_class = "App", resource_path = NULL)
```

Arguments

`root`	a component used as the top-level node for the shiny app
`id`	TODO
`app_class`	the html class attribute for the app wrapper
`resource_path`	path to a resource folder, if `NULL` styles will be inlined

Value

`shiny::tagList` object

See Also

Other compilation: `rsx_app()`, `rsx_server()`

Index