Package 'shinyjs'

January 13, 2020

Title Easily Improve the User Experience of Your Shiny Apps in Seconds

Version 1.1

Description Perform common useful JavaScript operations in Shiny apps that will greatly improve your apps without having to know any JavaScript. Examples include: hiding an element, disabling an input, resetting an input back to its original value, delaying code execution by a few seconds, and many more useful functions for both the end user and the developer. 'shinyjs' can also be used to easily call your own custom JavaScript functions from R.

URL https://deanattali.com/shinyjs

BugReports https://github.com/daattali/shinyjs/issues

Depends R (>= 3.1.0)

Imports digest (>= 0.6.8), htmltools (>= 0.2.9), jsonlite, shiny (>= 1.0.0)

Suggests knitr (>= 1.7), rmarkdown, shinyAce, testthat (>= 0.9.1), V8 (>= 0.6)

License AGPL-3

SystemRequirements pandoc with https support

LazyData true

VignetteBuilder knitr

RoxygenNote 6.1.1

NeedsCompilation no

Author Dean Attali [aut, cre]

Maintainer Dean Attali <daattali@gmail.com>

Repository CRAN

Date/Publication 2020-01-13 06:40:03 UTC

R topics documented:

classFuncs . . . 2
click . . . 4
delay . . . 5
disabled . . . 6
extendShinyjs . . . 7
hidden . . . 12
html . . . 13
inlineCSS . . . 15
messageFuncs . . . 16
onevent . . . 17
reset . . . 19
runcode . . . 21
runExample . . . 22
runjs . . . 23
shinyjs . . . 24
showLog . . . 24
stateFuncs . . . 25
useShinyjs . . . 27
visibilityFuncs . . . 28

Index **31**

`classFuncs` *Add/remove CSS class*

Description

Add or remove a CSS class from an HTML element.

`addClass` adds a CSS class, `removeClass` removes a CSS class, `toggleClass` adds the class if it is not set and removes the class if it is already set.

`addCssClass`, `removeCssClass`, and `toggleCssClass` are synonyms that may be safer to use if you're working with S4 classes (since they don't mask any existing S4 functions).

If `condition` is given to `toggleClass`, that condition will be used to determine if to add or remove the class. The class will be added if the condition evaluates to `TRUE` and removed otherwise. If you find yourself writing code such as `if (test()) addClass(id,cl) else removeClass(id,cl)` then you can use `toggleClass` instead: `toggleClass(id,cl,test())`.

CSS is a simple way to describe how elements on a web page should be displayed (position, colour, size, etc.). You can learn the basics at W3Schools.

Usage

```
addClass(id = NULL, class = NULL, selector = NULL, asis = FALSE)

addCssClass(id = NULL, class = NULL, selector = NULL, asis = FALSE)

removeClass(id = NULL, class = NULL, selector = NULL, asis = FALSE)

removeCssClass(id = NULL, class = NULL, selector = NULL,
  asis = FALSE)

toggleClass(id = NULL, class = NULL, condition = NULL,
  selector = NULL, asis = FALSE)

toggleCssClass(id = NULL, class = NULL, condition = NULL,
  selector = NULL, asis = FALSE)
```

Arguments

id	The id of the element/Shiny tag
class	The CSS class to add/remove
selector	JQuery selector of the elements to target. Ignored if the id argument is given. For example, to add a certain class to all inputs with class x, use selector = "input.x"
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).
condition	An optional argument to toggleClass, see 'Details' below.

Note

If you use S4 classes, you should be aware of the fact that both S4 and shinyjs use the removeClass() function. This means that when using S4, it is recommended to use removeCssClass() from shinyjs, and to use methods::removeClass() for S4 object.

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample, inlineCSS,

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      # Add a CSS class for red text colour
      inlineCSS(list(.red = "background: red")),
```

```
      actionButton("btn", "Click me"),
      p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Change the following line for more examples
        toggleClass("element", "red")
      })
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  toggleClass(class = "red", id = "element")
  addClass("element", "red")
  removeClass("element", "red")

## End(Not run)

## toggleClass can be given an optional `condition` argument, which
## determines if to add or remove the class
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      inlineCSS(list(.red = "background: red")),
      checkboxInput("checkbox", "Make it red"),
      p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observe({
        toggleClass(id = "element", class = "red",
                    condition = input$checkbox)
      })
    }
  )
}
```

`click` *Click on a Shiny button*

Description

The `click()` function can be used to programatically simulate a click on a Shiny `actionButton()`.

Usage

```
click(id, asis = FALSE)
```

Arguments

id	The id of the button
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      "Count:", textOutput("number", inline = TRUE), br(),
      actionButton("btn", "Click me"), br(),
      "The button will be pressed automatically every 3 seconds"
    ),
    server = function(input, output) {
      output$number <- renderText({
        input$btn
      })
      observe({
        click("btn")
        invalidateLater(3000)
      })
    }
  )
}
```

---

delay *Execute R code after a specified number of milliseconds has elapsed*

Description

You can use delay if you want to wait a specific amount of time before running code. This function can be used in combination with other shinyjs functions, such as hiding or resetting an element in a few seconds, but it can also be used with any code as long as it's used inside a Shiny app.

Usage

```
delay(ms, expr)
```

Arguments

ms	The number of milliseconds to wait (1000 milliseconds = 1 second) before running the expression.
expr	The R expression to run after the specified number of milliseconds has elapsed.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      p(id = "text", "This text will disappear after 3 seconds"),
      actionButton("close", "Close the app in half a second")
    ),
    server = function(input, output) {
      delay(3000, hide("text"))
      observeEvent(input$close, {
        delay(500, stopApp())
      })
    }
  )
}
```

---

disabled *Initialize a Shiny input as disabled*

Description

Create a Shiny input that is disabled when the Shiny app starts. The input can be enabled later with shinyjs::toggleState or shinyjs::enable.

Usage

```
disabled(...)
```

Arguments

...	Shiny input (or tagList or list of of tags that include inputs) to disable.

Value

The tag (or tags) that was given as an argument in a disabled state.

Note

`shinyjs` must be initialized with a call to `useShinyjs()` in the app's ui.

See Also

`useShinyjs`, `toggleState`, `enable`, `disable`

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      disabled(
        textInput("element", NULL, "I was born disabled")
      )
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        enable("element")
      })
    }
  )
}

library(shiny)
disabled(numericInput("num", NULL, 5), dateInput("date", NULL))
```

`extendShinyjs` *Extend shinyjs by calling your own JavaScript functions*

Description

Add your own JavaScript functions that can be called from R as if they were regular R functions. This is a more advanced technique and can only be used if you know JavaScript. See 'Basic Usage' below for more information or view the shinyjs webpage to learn more.

Usage

```
extendShinyjs(script, text, functions)
```

Arguments

`script`	Path to a JavaScript file that contains all the functions. Each function name must begin with 'shinyjs.', for example 'shinyjs.myfunc'. See 'Basic Usage' below.
`text`	Inline JavaScript code to use. If your JavaScript function is very short and you don't want to create a separate file for it, you can provide the code as a string. See 'Basic Usage' below.
`functions`	The names of the shinyjs JavaScript functions which you defined and want to be able to call using `shinyjs`. Only use this argument if you cannot install `V8` on your machine. I repeat: do not use this argument if you're able to install `V8` on your machine. For example, if you defined JavaScript functions named `shinyjs.foo` and `shinyjs.bar`, then use `functions = c("foo","bar")`.

Value

Scripts that `shinyjs` requires in order to run your JavaScript functions as if they were R code.

Basic Usage

Any JavaScript function defined in your script that begins with 'shinyjs.' will be available to run from R through the 'js$' variable. For example, if you write a JavaScript function called 'shinyjs.myfunc', then you can call it in R with 'js$myfunc()'.

It's recommended to write JavaScript code in a separate file and provide the filename as the `script` argument, but it's also possible to use the `text` argument to provide a string containing valid JavaScript code. Using the `text` argument is meant to be used when your JavaScript code is very short and simple.

As a simple example, here is a basic example of using `extendShinyjs` to define a function that changes the colour of the page.

```
library(shiny)
library(shinyjs)

jsCode <- "shinyjs.pageCol = function(params){$('body').css('background', params);}"

shinyApp(
  ui = fluidPage(
    useShinyjs(),
    extendShinyjs(text = jsCode),
    selectInput("col", "Colour:",
                c("white", "yellow", "red", "blue", "purple"))
  ),
  server = function(input, output) {
    observeEvent(input$col, {
      js$pageCol(input$col)
    })
  }
)
```

As the example above shows, after defining the JavaScript function `shinyjs.pageCol` and passing it to `extendShinyjs`, it's possible to call `js$pageCol()`.

You can add more functions to the JavaScript code, but remember that every function you want to use in R has to have a name beginning with 'shinyjs.'. See the section on passing arguments and the examples below for more information on how to write effective functions.

Running JavaScript code on page load

If there is any JavaScript code that you want to run immediately when the page loads rather than having to call it from the server, you can place it inside a `shinyjs.init` function. The function `shinyjs.init` will automatically be called when the Shiny app's HTML is initialized. A common use for this is when registering event handlers or initializing JavaScript objects, as these usually just need to run once when the page loads.

For example, the following example uses `shinyjs.init` to register an event handler so that every keypress will print its corresponding key code:

```
jscode <- "
shinyjs.init = function() {
  $(document).keypress(function(e) { alert('Key pressed: ' + e.which); });
}"
shinyApp(
  ui = fluidPage(
    useShinyjs(),
    extendShinyjs(text = jscode),
    "Press any key"
  ),
  server = function(input, output) {}
)
```

Passing arguments from R to JavaScript

Any `shinyjs` function that is called will pass a single array-like parameter to its corresponding JavaScript function. If the function in R was called with unnamed arguments, then it will pass an Array of the arguments; if the R arguments are named then it will pass an Object with key-value pairs.

For example, calling `js$foo("bar",5)` in R will call `shinyjs.foo(["bar",5])` in JS, while calling `js$foo(num = 5,id = "bar")` in R will call `shinyjs.foo({num : 5,id : "bar"})` in JS. This means that the `shinyjs.foo` function needs to be able to deal with both types of parameters.

To assist in normalizing the parameters, `shinyjs` provides a `shinyjs.getParams()` function which serves two purposes. First of all, it ensures that all arguments are named (even if the R function was called without names). Secondly, it allows you to define default values for arguments.

Here is an example of a JS function that changes the background colour of an element and uses `shinyjs.getParams()`.

```
shinyjs.backgroundCol = function(params) {
  var defaultParams = {
    id : null,
```

```
    col : "red"
  };
  params = shinyjs.getParams(params, defaultParams);

  var el = $("#" + params.id);
  el.css("background-color", params.col);
}
```

Note the `defaultParams` object that was defined and the call to `shinyjs.getParams`. It ensures that calling `js$backgroundCol("test","blue")` and `js$backgroundCol(id = "test",col = "blue")` and `js$backgroundCol(col = "blue",id = "test")` are all equivalent, and that if the colour parameter is not provided then "red" will be the default.

All the functions provided in `shinyjs` make use of `shinyjs.getParams`, and it is highly recommended to always use it in your functions as well. Notice that the order of the arguments in `defaultParams` in the JavaScript function matches the order of the arguments when calling the function in R with unnamed arguments.

See the examples below for a shiny app that uses this JS function.

Note

You still need to call `useShinyjs()` as usual, and the call to `useShinyjs()` must come before the call to `extendShinyjs()`.

The `V8` package is strongly recommended if you use this function.

If you are deploying your app to shinyapps.io and are using `extendShinyjs()`, then you need to let shinyapps.io know that the `V8` package is required. The easiest way to do this is by simply including `library(V8)` somewhere. This is an issue with shinyapps.io that might be resolved by them in the future – see here for more details.

See Also

`runExample`

Examples

```
## Not run:
  Example 1:
  Change the page background to a certain colour when a button is clicked.

    jsCode <- "shinyjs.pageCol = function(params){$('body').css('background', params);}"

    shinyApp(
      ui = fluidPage(
        useShinyjs(),
        extendShinyjs(text = jsCode),
        selectInput("col", "Colour:",
                    c("white", "yellow", "red", "blue", "purple"))
      ),
      server = function(input, output) {
        observeEvent(input$col, {
```

```
        js$pageCol(input$col)
      })
    }
  )

  # If you do not have `V8` package installed, you will need to add another
  # argument to the `extendShinyjs()` function:
  # extendShinyjs(text = jsCode, functions = c("pageCol"))

==============

Example 2:
Change the background colour of an element, using "red" as default

  jsCode <- '
  shinyjs.backgroundCol = function(params) {
    var defaultParams = {
      id : null,
      col : "red"
    };
    params = shinyjs.getParams(params, defaultParams);

    var el = $("#" + params.id);
    el.css("background-color", params.col);
  }'

  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      extendShinyjs(text = jsCode),
      p(id = "name", "My name is Dean"),
      p(id = "sport", "I like soccer"),
      selectInput("col", "Colour:",
                  c("white", "yellow", "red", "blue", "purple")),
      textInput("selector", "Element", "sport"),
      actionButton("btn", "Go")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        js$backgroundCol(input$selector, input$col)
      })
    }
  )

==============

Example 3:
Create an `increment` function that increments the number inside an HTML
tag (increment by 1 by default, with an optional parameter). Use a separate
file instead of providing the JS code in a string.

Create a JavaScript file "myfuncs.js":
  shinyjs.increment = function(params) {
```

```
    var defaultParams = {
      id : null,
      num : 1
    };
    params = shinyjs.getParams(params, defaultParams);

    var el = $("#" + params.id);
    el.text(parseInt(el.text()) + params.num);
  }
```

And a shiny app that uses the custom function we just defined. Note how the arguments can be either passed as named or unnamed, and how default values are set if no value is given to a parameter.

```
    library(shiny)
    shinyApp(
      ui = fluidPage(
        useShinyjs(),
        extendShinyjs("myfuncs.js"),
        p(id = "number", 0),
        actionButton("add", "js$increment('number')"),
        actionButton("add5", "js$increment('number', 5)"),
        actionButton("add10", "js$increment(num = 10, id = 'number')")
      ),
      server = function(input, output) {
        observeEvent(input$add, {
          js$increment('number')
        })
        observeEvent(input$add5, {
          js$increment('number', 5)
        })
        observeEvent(input$add10, {
          js$increment(num = 10, id = 'number')
        })
      }
    )

## End(Not run)
```

`hidden` *Initialize a Shiny tag as hidden*

Description

Create a Shiny tag that is invisible when the Shiny app starts. The tag can be made visible later with `shinyjs::toggle` or `shinyjs::show`.

Usage

```
hidden(...)
```

Arguments

...	Shiny tag (or tagList or list of of tags) to make invisible

Value

The tag (or tags) that was given as an argument in a hidden state.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, toggle, show, hide

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      hidden(
        p(id = "element", "I was born invisible")
      )
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        show("element")
      })
    }
  )
}

library(shiny)
hidden(span(id = "a"), div(id = "b"))
hidden(tagList(span(id = "a"), div(id = "b")))
hidden(list(span(id = "a"), div(id = "b")))
```

---

html *Change the HTML (or text) inside an element*

Description

Change the text or HTML inside an element. The given HTML can be any R expression, and it can either be appended to the currentcontents of the element or overwrite it (default).

Usage

```
html(id = NULL, html = NULL, add = FALSE, selector = NULL,
  asis = FALSE)
```

Arguments

id	The id of the element/Shiny tag
html	The HTML/text to place inside the element. Can be either simple plain text or valid HTML code.
add	If TRUE, then append html to the contents of the element; otherwise overwrite it.
selector	JQuery selector of the elements to target. Ignored if the id argument is given.
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Change the following line for more examples
        html("element", paste0("The date is ", date()))
      })
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  html("element", "Hello!")
  html("element", " Hello!", TRUE)
  html("element", "<strong>bold</strong> that was achieved with HTML")
  local({val <- "some text"; html("element", val)})
  html(id = "element", add = TRUE, html = input$btn)
```

```
## End(Not run)
```

`inlineCSS` *Add inline CSS*

Description

Add inline CSS to a Shiny app. This is simply a convenience function that gets called from a Shiny app's UI to make it less tedious to add inline CSS. If there are many CSS rules, it is recommended to use an external stylesheet.

CSS is a simple way to describe how elements on a web page should be displayed (position, colour, size, etc.). You can learn the basics at W3Schools.

Usage

```
inlineCSS(rules)
```

Arguments

`rules`	The CSS rules to add. Can either be a string with valid CSS code, or a named list of the form `list(selector = declarations)`, where `selector` is a valid CSS selector and `declarations` is a string or vector of declarations. See examples for clarification.

Value

Inline CSS code that is automatically inserted to the app's <head> tag.

Examples

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

  # Method 1 - passing a string of valid CSS
  shinyApp(
    ui = fluidPage(
      inlineCSS("#big { font-size:30px; }
                .red { color: red; border: 1px solid black;}"),
      p(id = "big", "This will be big"),
      p(class = "red", "This will be red and bordered")
    ),
    server = function(input, output) {}
  )

  # Method 2 - passing a list of CSS selectors/declarations
  # where each declaration is a full declaration block
  shinyApp(
    ui = fluidPage(
```

```
    inlineCSS(list(
      "#big" = "font-size:30px",
      ".red" = "color: red; border: 1px solid black;"
    )),
    p(id = "big", "This will be big"),
    p(class = "red", "This will be red and bordered")
  ),
  server = function(input, output) {}
)

# Method 3 - passing a list of CSS selectors/declarations
# where each declaration is a vector of declarations
shinyApp(
  ui = fluidPage(
    inlineCSS(list(
      "#big" = "font-size:30px",
      ".red" = c("color: red", "border: 1px solid black")
    )),
    p(id = "big", "This will be big"),
    p(class = "red", "This will be red and bordered")
  ),
  server = function(input, output) {}
)
}
```

`messageFuncs` *Show a message*

Description

`alert` (and its alias `info`) shows a message to the user as a simple popup.

`logjs` writes a message to the JavaScript console. `logjs` is mainly used for debugging purposes as a way to non-intrusively print messages, but it is also visible to the user if they choose to inspect the console. You can also use the `showLog` function to print the JavaScript message directly to the R console.

Usage

```
alert(text)

info(text)

logjs(text)
```

Arguments

`text`	The message to show. Can be either simple text or an R object.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample, showLog

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Change the following line for more examples
        alert(paste0("The date is ", date()))
      })
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  alert("Hello!")
  alert(text = R.Version())
  logjs(R.Version())

## End(Not run)
```

---

onevent *Run R code when an event is triggered on an element*

Description

onclick runs an R expression (either a shinyjs function or any other code) when an element is clicked.

onevent is similar, but can be used when any event is triggered on the element, not only a mouse click. See below for a list of possible event types. Using "click" results in the same behaviour as calling onclick.

Usage

```
onclick(id, expr, add = FALSE, asis = FALSE)

onevent(event, id, expr, add = FALSE, properties = NULL,
  asis = FALSE)
```

Arguments

id	The id of the element/Shiny tag
expr	The R expression or function to run after the event is triggered. If a function with an argument is provided, it will be called with the JavaScript Event properties as its argument. Using a function can be useful when you want to know, for example, what key was pressed on a "keypress" event or the mouse coordinates in a mouse event. See below for a list of properties.
add	If TRUE, then add expr to be executed after any other code that was previously set using onevent or onclick; otherwise expr will overwrite any previous expressions. Note that this parameter works well in web browsers but is buggy when using the RStudio Viewer.
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).
event	The event that needs to be triggered to run the code. See below for a list of event types.
properties	A list of JavaScript Event properties that should be available to the argument of the expr function. See below for more information about Event properties.

Event types

Any standard mouse or keyboard events that are supported by JQuery can be used. The standard list of events that can be used is: click, dblclick, hover, mousedown, mouseenter, mouseleave, mousemove, mouseout, mouseover, mouseup, keydown, keypress, keyup. You can also use any other non standard events that your browser supports or with the use of plugins (for example, there is a mousewheel plugin that you can use to listen to mousewheel events).

Event properties

If a function is provided to expr, the function will receive a list of JavaScript Event properties describing the current event as an argument. Different properties are available for different event types. The full list of porperties that can be returned is: altKey, button, buttons, clientX, clientY, ctrlKey, pageX, pageY, screenX, screenY, shiftKey, which, charCode, key, keyCode, offsetX, offsetY. If you want to retrieve any additional properties that are available in JavaScript for your event type, you can use the properties parameter.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      p(id = "date", "Click me to see the date"),
      p(id = "coords", "Click me to see the mouse coordinates"),
      p(id = "disappear", "Move your mouse here to make the text below disappear"),
      p(id = "text", "Hello")
    ),
    server = function(input, output) {
      onclick("date", alert(date()))
      onclick("coords", function(event) { alert(event) })
      onevent("mouseenter", "disappear", hide("text"))
      onevent("mouseleave", "disappear", show("text"))
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  onclick("disappear", toggle("text"))
  onclick(expr = text("date", date()), id = "date")

## End(Not run)
```

---

reset *Reset input elements to their original values*

Description

Reset any input element back to its original value. You can either reset one specific input at a time by providing the id of a shiny input, or reset all inputs within an HTML tag by providing the id of an HTML tag.

Reset can be performed on any traditional Shiny input widget, which includes: textInput, numericInput, sliderInput, selectInput, selectizeInput, radioButtons, dateInput, dateRangeInput, checkboxInput, checkboxGroupInput, colourInput, passwordInput, textAreaInput. Note that `actionButton` is not supported, meaning that you cannot reset the value of a button back to 0.

Usage

```
reset(id)
```

Arguments

id	The id of the input element to reset or the id of an HTML tag to reset all input elements inside it.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      div(
        id = "form",
        textInput("name", "Name", "Dean"),
        radioButtons("gender", "Gender", c("Male", "Female")),
        selectInput("letter", "Favourite letter", LETTERS)
      ),
      actionButton("resetAll", "Reset all"),
      actionButton("resetName", "Reset name"),
      actionButton("resetGender", "Reset Gender"),
      actionButton("resetLetter", "Reset letter")
    ),
    server = function(input, output) {
      observeEvent(input$resetName, {
        reset("name")
      })
      observeEvent(input$resetGender, {
        reset("gender")
      })
      observeEvent(input$resetLetter, {
        reset("letter")
      })
      observeEvent(input$resetAll, {
        reset("form")
      })
    }
  )
}
```

runcode *Construct to let you run arbitrary R code live in a Shiny app*

Description

Sometimes when developing a Shiny app, it's useful to be able to run some R code on-demand. This construct provides your app with a text input where you can enter any R code and run it immediately.

This can be useful for testing and while developing an app locally, but it **should not be included in an app that is accessible to other people**, as letting others run arbitrary R code can open you up to security attacks.

To use this construct, you must add a call to `runcodeUI()` in the UI of your app, and a call to `runcodeServer()` in the server function. You also need to initialize shinyjs with a call to `useShinyjs()` in the UI.

Usage

```
runcodeUI(code = "", type = c("text", "textarea", "ace"),
  width = NULL, height = NULL, includeShinyjs = NULL, id = NULL)

runcodeServer()
```

Arguments

`code`	The initial R code to show in the text input when the app loads
`type`	One of `"text"` (default), `"textarea"`, or `"ace"`. When using a text input, the R code will be limited to be typed within a single line, and is the recommended option. Textarea should be used if you want to write long multi-line R code. Note that you can run multiple expressions even in a single line by appending each R expression with a semicolon. Use of the `"ace"` option requires the `shinyAce` package.
`width`	The width of the editable code input (ignored when `type="ace"`)
`height`	The height of the editable code input (ignored when `type="text"`)
`includeShinyjs`	Deprecated. You should always make sure to initialize shinyjs using `useShinyjs`.
`id`	When used inside a shiny module, the module's id needs to be provided to `runcodeUI`. This argument should remain `NULL` when not used inside a module.

Note

You can only have one `runcode` construct in your shiny app. Calling this function multiple times within the same app will result in unpredictable behaviour.

See Also

`useShinyjs`

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      runcodeUI(code = "shinyjs::alert('Hello!')")
    ),
    server = function(input, output) {
      runcodeServer()
    }
  )
}
```

`runExample` *Run shinyjs examples*

Description

Launch a `shinyjs` example Shiny app that shows how to easily use `shinyjs` in an app.

Run without any arguments to see a list of available example apps. The "demo" example is also available online to experiment with.

Usage

```
runExample(example)
```

Arguments

`example`	The app to launch

Examples

```
## Only run this example in interactive R sessions
if (interactive()) {
  # List all available example apps
  runExample()

  runExample("sandbox")
  runExample("demo")
}
```

runjs *Run JavaScript code*

Description

Run arbitrary JavaScript code.

Usage

```
runjs(code)
```

Arguments

code	JavaScript code to run.

Note

shinyjs must be initialized with a call to `useShinyjs()` in the app's ui.

See Also

useShinyjs

Examples

```
if (interactive()) {
  library(shiny)
  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Run JS code that simply shows a message
        runjs("var today = new Date(); alert(today);")
      })
    }
  )
}
```

shinyjs *shinyjs*

Description

Easily improve the user experience of your Shiny apps in seconds

Details

`shinyjs` lets you perform common JavaScript operations that enhance the user experience in applications without having to know any JavaScript. Examples include: hiding an element, disabling an input, resetting an input back to its original value, delaying code execution by a few seconds, and many more useful functions. `shinyjs` also includes a colour picker widget, a colour picker RStudio addin, and can also be used to easily run your own custom JavaScript functions from R.

View the shinyjs website for more details and to see a demo.

showLog *Print any JavaScript console.log messages in the R console*

Description

When developing and debugging a Shiny that uses custom JavaScript code, it can be helpful to use `console.log()` messages in JavaScript. This function allows you to see these messages printed in the R console directly rather than having to open the JavaScript console in the browser to view the messages.

This function must be called in a Shiny app's server function, and you also need to pass the `showLog=TRUE` parameter to `useShinyjs()`.

Usage

```
showLog()
```

Note

Due to an issue in shiny (see https://github.com/rstudio/shiny/issues/928), duplicated consecutive log messages will not get printed in R.

Log messages that cannot be serialized in JavaScript (such as many JavaScript Event objects that are cyclic) will not be printed in R.

See Also

`logjs`

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      textInput("text", "Type something")
    ),
    server = function(input, output) {
      showLog()
      logjs("App started")
      observe({
        logjs(paste("Length of text:", nchar(input$text)))
      })
    }
  )
}
```

`stateFuncs` *Enable/disable an input element*

Description

Enable or disable an input element. A disabled element is not usable and not clickable, while an enabled element (default) can receive user input. Any shiny input tag can be used with these functions.

`enable` enables an input, `disable` disabled an input,`toggleState` enables an input if it is disabled and disables an input if it is already enabled.

If `condition` is given to `toggleState`, that condition will be used to determine if to enable or disable the input. The element will be enabled if the condition evaluates to `TRUE` and disabled otherwise. If you find yourself writing code such as `if (test()) enable(id) else disable(id)` then you can use `toggleState` instead: `toggleState(id,test())`.

Usage

```
enable(id = NULL, selector = NULL, asis = FALSE)

disable(id = NULL, selector = NULL, asis = FALSE)

toggleState(id = NULL, condition = NULL, selector = NULL,
  asis = FALSE)
```

Arguments

`id`	The id of the input element/Shiny tag

selector	Query selector of the elements to target. Ignored if the id argument is given. For example, to disable all text inputs, use selector = "input[type='text']"
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).
condition	An optional argument to toggleState. The element will be enabled when the condition is TRUE, and disabled otherwise.

Note

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample disabled

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      textInput("element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Change the following line for more examples
        toggleState("element")
      })
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  toggleState(id = "element")
  enable("element")
  disable("element")

  # Similarly, the "element" text input can be changed to many other
  # input tags, such as the following examples
  actionButton("element", "I'm a button")
  fileInput("element", "Choose a file")
  selectInput("element", "I'm a select box", 1:10)

## End(Not run)

## toggleState can be given an optional `condition` argument, which
## determines if to enable or disable the input
```

```
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      textInput("text", "Please type at least 3 characters"),
      actionButton("element", "Submit")
    ),
    server = function(input, output) {
      observe({
        toggleState(id = "element", condition = nchar(input$text) >= 3)
      })
    }
  )
}
```

`useShinyjs` *Set up a Shiny app to use shinyjs*

Description

This function must be called from a Shiny app's UI in order for all other `shinyjs` functions to work.

You can call `useShinyjs()` from anywhere inside the UI, as long as the final app UI contains the result of `useShinyjs()`.

Usage

```
useShinyjs(rmd = FALSE, debug = FALSE, html = FALSE,
  showLog = NULL)
```

Arguments

`rmd`	Set this to `TRUE` only if you are using `shinyjs` inside an interactive R markdown document. If using this option, view the README online to learn how to use shinyjs in R markdown documents.
`debug`	Set this to `TRUE` if you want to see detailed debugging statements in the JavaScript console. Can be useful when filing bug reports to get more information about what is going on.
`html`	Set this to `TRUE` only if you are using `shinyjs` in a Shiny app that builds the entire user interface with a custom HTML file. If using this option, view the README online to learn how to use shinyjs in these apps.
`showLog`	Deprecated.

Details

If you're a package author and including `shinyjs` in a function in your your package, you need to make sure `useShinyjs()` is called either by the end user's Shiny app or by your function's UI.

Value

Scripts that shinyjs requires that are automatically inserted to the app's <head> tag. A side effect of calling this function is that a shinyjs directory is added as a resource path using addResourcePath.

See Also

runExample extendShinyjs

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      textInput("element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Run a simply shinyjs function
        toggle("element")
      })
    }
  )
}
```

---

visibilityFuncs *Display/hide an element*

Description

Display or hide an HTML element.

show makes an element visible, hide makes an element invisible, toggle displays the element if it it hidden and hides it if it is visible.

showElement, hideElement, and toggleElement are synonyms that may be safer to use if you're working with S4 classes (since they don't mask any existing S4 functions).

If condition is given to toggle, that condition will be used to determine if to show or hide the element. The element will be shown if the condition evaluates to TRUE and hidden otherwise. If you find yourself writing code such as if (test()) show(id) else hide(id) then you can use toggle instead: toggle(id = id, condition = test()).

Usage

```
show(id = NULL, anim = FALSE, animType = "slide", time = 0.5,
  selector = NULL, asis = FALSE)

showElement(id = NULL, anim = FALSE, animType = "slide",
  time = 0.5, selector = NULL, asis = FALSE)

hide(id = NULL, anim = FALSE, animType = "slide", time = 0.5,
  selector = NULL, asis = FALSE)

hideElement(id = NULL, anim = FALSE, animType = "slide",
  time = 0.5, selector = NULL, asis = FALSE)

toggle(id = NULL, anim = FALSE, animType = "slide", time = 0.5,
  selector = NULL, condition = NULL, asis = FALSE)

toggleElement(id = NULL, anim = FALSE, animType = "slide",
  time = 0.5, selector = NULL, condition = NULL, asis = FALSE)
```

Arguments

id	The id of the element/Shiny tag
anim	If TRUE then animate the behaviour
animType	The type of animation to use, either "slide" or "fade"
time	The number of seconds to make the animation last
selector	JQuery selector of the elements to show/hide. Ignored if the id argument is given. For example, to select all span elements with class x, use selector = "span.x"
asis	If TRUE, use the ID as-is even when inside a module (instead of adding the namespace prefix to the ID).
condition	An optional argument to toggle, see 'Details' below.

Details

If you want to hide/show an element in a few seconds rather than immediately, you can use the delay function.

Note

If you use S4 classes, you should be aware of the fact that both S4 and shinyjs use the show() function. This means that when using S4, it is recommended to use showElement() from shinyjs, and to use methods::show() for S4 object.

shinyjs must be initialized with a call to useShinyjs() in the app's ui.

See Also

useShinyjs, runExample, hidden, delay

Examples

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

  shinyApp(
    ui = fluidPage(
      useShinyjs(),  # Set up shinyjs
      actionButton("btn", "Click me"),
      textInput("text", "Text")
    ),
    server = function(input, output) {
      observeEvent(input$btn, {
        # Change the following line for more examples
        toggle("text")
      })
    }
  )
}
## Not run:
  # The shinyjs function call in the above app can be replaced by
  # any of the following examples to produce similar Shiny apps
  toggle(id = "text")
  delay(1000, toggle(id = "text")) # toggle in 1 second
  toggle("text", TRUE)
  toggle("text", TRUE, "fade", 2)
  toggle(id = "text", time = 1, anim = TRUE, animType = "slide")
  show("text")
  show(id = "text", anim = TRUE)
  hide("text")
  hide(id = "text", anim = TRUE)

## End(Not run)

## toggle can be given an optional `condition` argument, which
## determines if to show or hide the element
if (interactive()) {
  shinyApp(
    ui = fluidPage(
      useShinyjs(),
      checkboxInput("checkbox", "Show the text", TRUE),
      p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      observe({
        toggle(id = "element", condition = input$checkbox)
      })
    }
  )
}
```

Index

addClass *(*classFuncs*)*, 2
addCssClass *(*classFuncs*)*, 2
addResourcePath, *28*
alert *(*messageFuncs*)*, 16

classFuncs, 2
click, 4

delay, 5, *29*
disable, *7*
disable *(*stateFuncs*)*, 25
disabled, 6, *26*

enable, *7*
enable *(*stateFuncs*)*, 25
extendShinyjs, 7, *28*

hidden, 12, *29*
hide, *13*
hide *(*visibilityFuncs*)*, 28
hideElement *(*visibilityFuncs*)*, 28
html, 13

info *(*messageFuncs*)*, 16
inlineCSS, *3*, 15

logjs, *24*
logjs *(*messageFuncs*)*, 16

messageFuncs, 16

onclick *(*onevent*)*, 17
onevent, 17

removeClass *(*classFuncs*)*, 2
removeCssClass *(*classFuncs*)*, 2
reset, 19
runcode, 21
runcodeServer *(*runcode*)*, 21
runcodeUI *(*runcode*)*, 21
runExample, *3*, *5*, *6*, *10*, *14*, *17*, *19*, *20*, 22, *26*, *28*, *29*

runjs, 23

shinyjs, 24
shinyjs-package *(*shinyjs*)*, 24
show, *13*
show *(*visibilityFuncs*)*, 28
showElement *(*visibilityFuncs*)*, 28
showLog, *16*, *17*, 24
stateFuncs, 25

toggle, *13*
toggle *(*visibilityFuncs*)*, 28
toggleClass *(*classFuncs*)*, 2
toggleCssClass *(*classFuncs*)*, 2
toggleElement *(*visibilityFuncs*)*, 28
toggleState, *7*
toggleState *(*stateFuncs*)*, 25

useShinyjs, *3*, *5–7*, *13*, *14*, *17*, *19–21*, *23*, *26*, 27, *29*

visibilityFuncs, 28