Package 'shinyjs'

November 8, 2015

Title Perform Common JavaScript Operations in Shiny Apps using Plain R Code

Version 0.2.3

Description Perform common JavaScript operations in Shiny 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' can also be used to easily run your own custom JavaScript functions from R.

URL https://github.com/daattali/shinyjs

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

Depends R (>= 3.1.0)

Imports digest (>= 0.6.8), htmltools (>= 0.2.6), shiny (>= 0.11.1), stats

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

License MIT + file LICENSE

LazyData true

VignetteBuilder knitr

RoxygenNote 5.0.0

NeedsCompilation no

Author Dean Attali [aut, cre]

Maintainer Dean Attali <daattali@gmail.com>

Repository CRAN

Date/Publication 2015-11-08 08:40:39

R topics documented:

`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.

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 evalutes 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(...)

removeClass(...)

toggleClass(...)
```

Arguments

`...`	The following parameters are available:
`id`	The id of the element/Shiny tag

class	The CSS class to add/remove
condition	An optional argument to toggleClass, see 'Details' below.
selector	JQuery selector of the elements to target. Ignored if the id argument is given. For example, to add a certain clas

Note

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

See Also

useShinyjs, runExample, inlineCSS,

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      # Add a CSS class for red text colour
      inlineCSS(list(.red = "background: red")),
      shiny::actionButton("btn", "Click me"),
      shiny::p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      shiny::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()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),
      inlineCSS(list(.red = "background: red")),
      shiny::checkboxInput("checkbox", "Make it red"),
      shiny::p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
```

```
      shiny::observe({
        toggleClass(id = "element", class = "red",
                    condition = input$checkbox)
      })
    }
  )
}
```

---

colourInput *Create a colour input control*

Description

Create an input control to select a colour.

Usage

```
colourInput(inputId, label, value = "white", showColour = c("both", "text",
  "background"), palette = c("square", "limited"), allowedCols,
  allowTransparent = FALSE, transparentText)
```

Arguments

inputId	The input slot that will be used to access the value.
label	Display label for the control, or 'NULL for no label.
value	Initial value (can be a colour name or HEX code)
showColour	Whether to show the chosen colour as text inside the input, as the background colour of the input, or both (default).
palette	The type of colour palette to allow the user to select colours from. square (default) shows a square colour palette that allows the user to choose any colour, while limited only gives the user a predefined list of colours to choose from.
allowedCols	A list of colours that the user can choose from. Only applicable when palette == "limited". The limited palette uses a default list of 40 colours if allowedCols is not defined.
allowTransparent	If TRUE, then add a checkbox that allows the user to select the transparent colour.
transparentText	The text to show beside the transparency checkbox when allowTransparent is TRUE. The default value is "Transparent", but you can change it to "None" or any other string. This has no effect on the return value from the input; when the checkbox is checked, the input will always return the string "transparent".

Details

A colour input allows users to select a colour by clicking on the desired colour, or by entering a valid HEX colour in the input box. The input can be initialized with either a colour name or a HEX value, but the value returned from the input will be an uppercase HEX value in both cases.

Since most functions in R that accept colours can also accept the value "transparent", `colourInput` has an option to allow selecting the "transparent" colour. When the user checks the checkbox for this special colour, the returned value form the input is "transpanrent", otherwise the return value is always a HEX value.

Note

Unlike the rest of the `shinyjs` functions, this function does not require you to call `useShinyjs()` first.

See http://daattali.com/shiny/colourInput/ for a live demo.

See Also

`updateColourInput`

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      shiny::strong("Selected colour:",
                    shiny::textOutput("value", inline = TRUE)),
      colourInput("col", "Choose colour", "red"),
      shiny::h3("Update colour input"),
      shiny::textInput("text", "New colour: (colour name or HEX value)"),
      shiny::selectInput("showColour", "Show colour",
        c("both", "text", "background")),
      shiny::selectInput("palette", "Colour palette",
        c("square", "limited")),
      shiny::checkboxInput("allowTransparent", "Allow transparent", FALSE),
      shiny::actionButton("btn", "Update")
    ),
    server = function(input, output, session) {
      shiny::observeEvent(input$btn, {
        updateColourInput(session, "col",
          value = input$text, showColour = input$showColour,
          allowTransparent = input$allowTransparent,
          palette = input$palette)
      })
      output$value <- shiny::renderText(input$col)
    }
  )
}
```

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

disabled(shiny::numericInput("num", NULL, 5), shiny::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 full README and demos to learn more.

Usage

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

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.

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);}"

runApp(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); });
}"
runApp(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$backgroundCol(col = "blue", id = "test") are all equivalent, and that if the colour parameter is not provided then "red" will be the default. All the functions provided in shinyjs make use of shinyjs.getParams, and it is highly recommended to always use it in your functions as well. Notice that the order of the arguments in defaultParams in the JavaScript function matches the order of the arguments when calling the function in R with unnamed arguments. This means that js$backgroundCol("blue", "test") will not work because the arguments are unnamed and the JS function expects the id to come before the colour. 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 required to 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);}"
```

```
  runApp(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)
      })
    }
  ))

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

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);
  }'

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

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

`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()) {
  # Method 1 - passing a string of valid CSS
  shiny::shinyApp(
    ui = shiny::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
  shiny::shinyApp(
    ui = shiny::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
  shiny::shinyApp(
    ui = shiny::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

`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.

Usage

```
info(...)

logjs(...)
```

Arguments

`...`	The following parameters are available:
`text`	The message to show. Can be either simple quoted text or an R variable or expression.

Note

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

See Also

useShinyjs, runExample

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      shiny::actionButton("btn", "Click me")
    ),
    server = function(input, output) {
      shiny::observeEvent(input$btn, {
        # Change the following line for more examples
        info(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
  info("Hello!")
  info(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)

onevent(event, id, expr, add = FALSE)
```

Arguments

id	The id of the element/Shiny tag
expr	The R expression to run after the event is triggered
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.
event	The event that needs to be triggered to run the code. See below for a list of possible event types.

Possible event types

Any mouse or keyboard events that are supported by JQuery can be used. The full list of events that can be used is: click, dblclick, hover, mousedown, mouseenter, mouseleave, mousemove, mouseout, mouseover, mouseup, keydown, keypress, keyup.

Note

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

See Also

useShinyjs, runExample

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      shiny::p(id = "date", "Click me to see the date"),
     shiny::p(id = "disappear", "Move your mouse here to make the text below disappear"),
      shiny::p(id = "text", "Hello")
    ),
    server = function(input, output) {
      onclick("date", info(date()))
      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. Buttons are not supported, meaning that you cannot use this function to reset the value of an action 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()) {
  runApp(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")
      })
    }
  ))
}
```

`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. This is mainly useful when developing and debugging a Shiny app, as it is generally considered dangerous to expose a way for end users to evaluate arbitrary code.

Usage

```
runjs(...)
```

Arguments

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

shinyjs *shinyjs*

Description

Perform common JavaScript operations in Shiny apps using plain R code.

Details

`shinyjs` lets you perform common JavaScript operations in Shiny 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` can also be used to easily run your own custom JavaScript functions from R.

View a demo Shiny app or see the full README on GitHub.

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 evalutes 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(...)

disable(...)

toggleState(...)
```

Arguments

`...`	The following parameters are available:
`id`	The id of the input element/Shiny tag
`condition`	An optional argument to `toggleState`, see 'Details' below.
`selector`	JQuery selector of the elements to target. Ignored if the `id` argument is given. For example, to disable all text inp

Note

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

See Also

useShinyjs, runExample disabled

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      shiny::actionButton("btn", "Click me"),
      shiny::textInput("element", "Watch what happens to me")
    ),
    server = function(input, output) {
      shiny::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
  shiny::actionButton("element", "I'm a button")
  shiny::fileInput("element", "Choose a file")
  shiny::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()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),
      shiny::textInput("text", "Please type at least 3 characters"),
      shiny::actionButton("element", "Submit")
    ),
    server = function(input, output) {
      shiny::observe({
        toggleState(id = "element", condition = nchar(input$text) >= 3)
      })
```

```
    }
  )
}
```

`text` *Change the text inside an element*

Description

Change the text or HTML inside an element. The given text can be any R expression, and it can either be appended to the contents of the element or overwrite it (default). This function can also be used to add HTML tags inside another element by passing in valid HTML instead of plain text.

Usage

```
text(...)
```

Arguments

`...`	The following parameters are available:
`id`	The id of the element/Shiny tag
`text`	The text to place inside the element. Can be either simple quoted text, R expressions, or valid HTML code.
`add`	If `TRUE`, then append `text` to the contents of the element; otherwise overwrite it. (default: `FALSE`)

Note

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

See Also

`useShinyjs`, `runExample`

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      shiny::actionButton("btn", "Click me"),
      shiny::p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      shiny::observeEvent(input$btn, {
        # Change the following line for more examples
        text("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
  text("element", "Hello!")
  text("element", " Hello!", TRUE)
  text("element", "<strong>bold</strong> that was achieved with HTML")
  local({val <- "some text"; text("element", val)})
  text(id = "element", add = TRUE, text = input$btn)

## End(Not run)
```

`updateColourInput` *Change the value of a colour input*

Description

Change the value of a colour input on the client.

Usage

```
updateColourInput(session, inputId, label = NULL, value = NULL,
  showColour = NULL, palette = NULL, allowedCols = NULL,
  allowTransparent = NULL, transparentText = NULL)
```

Arguments

`session`	The `session` object passed to function given to `shinyServer`.
`inputId`	The id of the colour input object.
`label`	The label to set for the input object.
`value`	The value to set for the input object.
`showColour`	Whether to shoW the chosen colour via text, background, or both.
`palette`	The type of colour palette to allow the user to select colours from.
`allowedCols`	A list of colours that the user can choose from.
`allowTransparent`	If `TRUE`, then add a checkbox that allows the user to select the `transparent` colour.
`transparentText`	The text to show beside the transparency checkbox when `allowTransparent` is `TRUE`

Details

The update function sends a message to the client, telling it to change the settings of a colour input object.
This function works similarly to the update functions provided by shiny.
Any argument with `NULL` values will be ignored.

Note

Unlike the rest of the shinyjs functions, this function does not require you to call useShinyjs() first.

See http://daattali.com/shiny/colourInput/ for a live demo.

See Also

colourInput

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      shiny::div("Selected colour:",
                 shiny::textOutput("value", inline = TRUE)),
      colourInput("col", "Choose colour", "red"),
      shiny::h3("Update colour input"),
      shiny::textInput("text", "New colour: (colour name or HEX value)"),
      shiny::selectInput("showColour", "Show colour",
        c("both", "text", "background")),
      shiny::checkboxInput("allowTransparent", "Allow transparent", FALSE),
      shiny::actionButton("btn", "Update")
    ),
    server = function(input, output, session) {
      shiny::observeEvent(input$btn, {
        updateColourInput(session, "col",
          value = input$text, showColour = input$showColour,
          allowTransparent = input$allowTransparent)
      })
      output$value <- shiny::renderText(input$col)
    }
  )
}
```

---

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.

Usage

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

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.

Value

Scripts that shinyjs requires that are automatically inserted to the app's <head> tag.

See Also

runExample extendShinyjs

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
      shiny::actionButton("btn", "Click me"),
      shiny::p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      shiny::observe({
        if (input$btn == 0) {
          return(NULL)
        }
        # 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.

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 evalutes 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(...)

hide(...)

toggle(...)
```

Arguments

`...`	The following parameters are available:
`id`	The id of the element/Shiny tag
`anim`	If `TRUE` then animate the behaviour (default: `FALSE`)
`animType`	The type of animation to use, either `"slide"` or `"fade"` (default: `"slide"`)
`time`	The number of seconds to make the animation last (default: `0.5`)
`selector`	JQuery selector of the elements to show/hide. Ignored if the `id` argument is given. For example, to select all spa
`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

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

See Also

`useShinyjs`, `runExample`, `hidden`, `delay`

Examples

```
if (interactive()) {
  shiny::shinyApp(
    ui = shiny::fluidPage(
      useShinyjs(),  # Set up shinyjs
```

```
      shiny::actionButton("btn", "Click me"),
      shiny::p(id = "element", "Watch what happens to me")
    ),
    server = function(input, output) {
      shiny::observeEvent(input$btn, {
        # Change the following line for more examples
        toggle("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
  toggle(id = "element")
  delay(1000, toggle(id = "element")) # toggle in 1 second
  toggle("element", TRUE)
  toggle("element", TRUE, "fade", 2)
  toggle(id = "element", time = 1, anim = TRUE, animType = "slide")
  show("element")
  show(id = "element", anim = TRUE)
  hide("element")
  hide(id = "element", anim = TRUE)

## End(Not run)

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

Index