Package: manipulateWidget (via r-universe)

September 2, 2024

Type Package

Title Add Even More Interactivity to Interactive Charts

Version 0.11.1

Description Like package 'manipulate' does for static graphics, this package helps to easily add controls like sliders, pickers, checkboxes, etc. that can be used to modify the input data or the parameters of an interactive chart created with package 'htmlwidgets'.

URL https://github.com/rte-antares-rpackage/manipulateWidget

License GPL (>= 2) | file LICENSE

Depends R (>= 2.10)

Imports shiny (>= 1.0.3), miniUI, htmltools, htmlwidgets, knitr, methods, tools, base64enc, grDevices, codetools, webshot, shinyjs

Suggests dygraphs, leaflet, plotly, xts, rmarkdown, testthat, covr

LazyData TRUE

RoxygenNote 7.1.1

VignetteBuilder knitr

Encoding UTF-8

Repository https://rte-antares-rpackage.r-universe.dev

RemoteUrl https://github.com/rte-antares-rpackage/manipulatewidget

RemoteRef HEAD

RemoteSha 3f9f6f2be4f53d509e1fc4f165a633d9dfe93cf4

Contents

`manipulateWidget-package`

Add even more interactivity to interactive charts

Description

This package is largely inspired by the `manipulate` package from Rstudio. It can be used to easily create graphical interface that let the user modify the data or the parameters of an interactive chart. It also provides the `combineWidgets` function to easily combine multiple interactive charts in a single view.

Details

`manipulateWidget` is the main function of the package. It accepts an expression that generates an interactive chart (and more precisely an `htmlwidget` object. See `http://www.htmlwidgets.org/` if you have never heard about it) and a set of controls created with functions `mwSlider`, `mwCheckbox`... which are used to dynamically change values within the expression. Each time the user modifies the value of a control, the expression is evaluated again and the chart is updated. Consider the following code:

```
manipulateWidget(myPlotFun(country), country = mwSelect(c("BE", "DE", "ES", "FR")))
```

It will generate a graphical interface with a select input on its left with options "BE", "DE", "ES", "FR". By default, at the beginning the value of the variable `country` will be equal to the first choice of the corresponding input. So the function will first execute `myPlotFun("BE")` and the result will

be displayed in the main panel of the interface. If the user changes the value to "FR", then the expression `myPlotFun("FR")` is evaluated and the new result is displayed.

The interface also contains a button "Done". When the user clicks on it, the last chart is returned. It can be stored in a variable, be modified by the user, saved as a html file with `saveWidget` from package `htmlwidgets` or converted to a static image file with package `webshot`.

Finally one can easily create complex layouts thanks to function `combineWidgets`. For instance, assume we want to see a map that displays values of some variable for a given year, but on its right side we also want to see the distributions of three variables. Then we could write:

```
myPlotFun <- function(year, variable) {
  combineWidgets(
    ncol = 2, colSize = c(3, 1),
    myMap(year, variable),
    combineWidgets(
      ncol = 1,
      myHist(year, "V1"),
      myHist(year, "V2"),
      myHist(year, "V3"),
    )
  )
}

manipulateWidget(
  myPlotFun(year, variable),
  year = mwSlider(2000, 2016, value = 2000),
  variable = mwSelect(c("V1", "V2", "V3"))
)
```

Of course, `combineWidgets` can be used outside of `manipulateWidget`. For instance, it can be used in an Rmarkdown document to easily put together interactive charts.

For more concrete examples of usage, you should look at the documentation and especially the examples of `manipulateWidget` and `combineWidgets`.

See Also

`manipulateWidget`, `combineWidgets`

`combineWidgets` *Combine several interactive plots*

Description

This function combines different htmlwidgets in a unique view.

Usage

```
combineWidgets(
  ...,
  list = NULL,
  nrow = NULL,
  ncol = NULL,
  title = NULL,
  rowsize = 1,
  colsize = 1,
  byrow = TRUE,
  titleCSS = "",
  header = NULL,
  footer = NULL,
  leftCol = NULL,
  rightCol = NULL,
  width = NULL,
  height = NULL
)
```

Arguments

`...`	htmlwidgets to combine. If this list contains objects that are not htmlwidgets, the function tries to convert them into a character string which is interpreted as html content.
`list`	Instead of directly passing htmlwidgets to the function, one can pass a list of htmlwidgets and objects coercible to character. In particular, it can be usefull if multiple htmlwidgets have been generated using a loop function like `lapply`.
`nrow`	Number of rows of the layout. If `NULL`, the function will automatically take a value such that are at least as many cells in the layout as the number of htmlwidgets.
`ncol`	Number of columns of the layout.If `NULL`, the function will automatically take a value such that are at least as many cells in the layout as the number of htmlwidgets.
`title`	Title of the view.
`rowsize`	This argument controls the relative size of each row. For instance, if the layout has two rows and `rowsize = c(2,1)`, then the width of the first row will be twice the one of the second one. This argument is recycled to fit the number of rows.
`colsize`	Same as rowsize but for the height of the columns of the layout.
`byrow`	If `TRUE`, then the layout is filled by row. Else it is filled by column.
`titleCSS`	A character containing css properties to modify the appearance of the title of the view.
`header`	Content to display between the title and the combined widgets. It can be a single character string or html tags.
`footer`	Content to display under the combined widgets. It can be a single character string or html tags.

leftCol	Content to display on the left of the combined widgets. It can be a single character string or html tags.
rightCol	Content to display on the right the combined widgets. It can be a single character string or html tags.
width	Total width of the layout (optional, defaults to automatic sizing).
height	Total height of the layout (optional, defaults to automatic sizing).

Details

The function only allows table like layout : each row has the same number of columns and reciprocally. But it is possible to create more complex layout by nesting combined htmlwidgets. (see examples)

Value

A htmlwidget object of class `combineWidget`. Individual widgets are stored in element `widgets` and can be extracted or updated. This is useful when a function returns a `combineWidgets` object but user wants to keep only one widget or to update one of them (see examples).

Examples

```
if (require(plotly)) {
  data(iris)

  combineWidgets(title = "The Iris dataset",
    plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
    plot_ly(iris, x = ~Sepal.Width, type = "histogram", nbinsx = 20),
    plot_ly(iris, x = ~Petal.Length, type = "histogram", nbinsx = 20),
    plot_ly(iris, x = ~Petal.Width, type = "histogram", nbinsx = 20)
  )

  # Create a more complex layout by nesting combinedWidgets
  combineWidgets(title = "The iris data set: sepals", ncol = 2, colsize = c(2,1),
    plot_ly(iris, x = ~Sepal.Length, y = ~Sepal.Width, type = "scatter",
            mode = "markers", color = ~Species),
    combineWidgets(
      plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
      plot_ly(iris, x = ~Sepal.Width, type = "histogram", nbinsx = 20)
    )
  )

  # combineWidgets can also be used on a single widget to easily add to it a
  # title and a footer.
  require(shiny)
  comments <- tags$div(
    "Wow this plot is so ",
    tags$span("amazing!!", style = "color:red;font-size:36px")
  )

  combineWidgets(
    plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
```

```
  title = "Distribution of Sepal Length",
  footer = comments
)

# It is also possible to combine htmlwidgets with text or other html elements
myComment <- tags$div(
  style="height:100%;background-color:#eee;padding:10px;box-sizing:border-box",
  tags$h2("Comment"),
  tags$hr(),
  "Here is a very clever comment about the awesome graphics you just saw."
)
combineWidgets(
  plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
  plot_ly(iris, x = ~Sepal.Width, type = "histogram", nbinsx = 20),
  plot_ly(iris, x = ~Petal.Length, type = "histogram", nbinsx = 20),
  myComment
)

# Updating individual widgets.
myWidget <- combineWidgets(
  plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
  plot_ly(iris, x = ~Sepal.Width, type = "histogram", nbinsx = 20),
  ncol = 2
)
myWidget


myWidget$widgets[[1]] <- myWidget$widgets[[1]] %>%
  layout(title = "Histogram of Sepal Length")

myWidget$widgets[[2]] <- myWidget$widgets[[2]] %>%
  layout(title = "Histogram of Sepal Width")

myWidget


# Instead of passing directly htmlwidgets to the function, one can pass
# a list containing htmlwidgets. This is especially useful when the widgets
# are generated using a loop function like "lapply" or "replicate".
#
# The following code generates a list of 12 histograms and use combineWidgets
# to display them.
samples <- replicate(12, plot_ly(x = rnorm(100), type = "histogram", nbinsx = 20),
                     simplify = FALSE)
combineWidgets(list = samples, title = "12 samples of the same distribution")
}
```

---

combineWidgets-shiny *Shiny bindings for combineWidgets*

Description

Output and render functions for using combineWidgets within Shiny applications and interactive Rmd documents.

Usage

```
combineWidgetsOutput(outputId, width = "100%", height = "400px")

renderCombineWidgets(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

outputId	output variable to read from
width, height	Must be a valid CSS unit (like '100%', '400px', 'auto') or a number, which will be coerced to a string and have 'px' appended.
expr	An expression that generates a combineWidgets
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.

---

compareOptions *Options for comparison mode*

Description

This function generates a list of options that are used by manipulateWidget to compare multiple charts.

Usage

```
compareOptions(ncharts = NULL, nrow = NULL, ncol = NULL, allowCompare = TRUE)
```

Arguments

ncharts	Number of charts to generate.
nrow	Number of rows. If NULL, the function tries to pick the best number of rows given the number of charts and columns.
ncol	Number of columns. If NULL, the function tries to pick the best number of columns given the number of charts and rows.
allowCompare	If TRUE (the default), then the user can use the UI to add or remove charts and choose which variables to compare

Value

List of options

Examples

```
if (require(dygraphs)) {

  mydata <- data.frame(
    year = 2000+1:100,
    series1 = rnorm(100),
    series2 = rnorm(100),
    series3 = rnorm(100)
  )
  manipulateWidget(
    dygraph(mydata[range[1]:range[2] - 2000, c("year", series)], main = title),
    range = mwSlider(2001, 2100, c(2001, 2100)),
    series = mwSelect(c("series1", "series2", "series3")),
    title = mwText("Fictive time series"),
    .compare = list(title = NULL, series = NULL),
    .compareOpts = compareOptions(ncharts = 4)
  )

  manipulateWidget(
    dygraph(mydata[range[1]:range[2] - 2000, c("year", series)], main = title),
    range = mwSlider(2001, 2100, c(2001, 2100)),
    series = mwSelect(c("series1", "series2", "series3")),
    title = mwText("Fictive time series"),
    .compare = list(title = NULL, series = NULL),
    .compareOpts = compareOptions(ncharts = 3, nrow = 3)
  )
}
```

---

knit_print.MWController

knit_print method for MWController object

Description

knit_print method for MWController object

Usage

```
knit_print.MWController(x, ...)
```

Arguments

x	MWController object
...	arguments passed to function knit_print

`manipulateWidget` *Add Controls to Interactive Plots*

Description

This function permits to add controls to an interactive plot created with packages like `dygraphs`, `highcharter` or `plotly` in order to change the input data or the parameters of the plot.

Technically, the function starts a shiny gadget. The R session is bloqued until the user clicks on "cancel" or "done". If he clicks on "done", then the the function returns the last displayed plot so the user can modify it and/or save it.

Usage

```
manipulateWidget(
  .expr,
  ...,
  .updateBtn = FALSE,
  .saveBtn = TRUE,
  .exportBtn = TRUE,
  .exportType = c("html2canvas", "webshot"),
  .viewer = c("pane", "window", "browser"),
  .compare = NULL,
  .compareOpts = compareOptions(),
  .translations = mwTranslations(),
  .return = function(widget, envs) {     widget },
  .width = NULL,
  .height = NULL,
  .runApp = TRUE
)
```

Arguments

`.expr`	expression to evaluate that returns an interactive plot of class `htmlwidget`. This expression is re-evaluated each time a control is modified.
`...`	One or more named control arguments created with functions `mwSlider`, `mwText`, etc. The name of each control is the name of the variable the controls modifies in the expression. One can also create a group of inputs by passing a list of such control arguments. for instance `mygroup = list(txt = mwText(""), nb = mwNumeric(0))` creates a group of inputs named mygroup with two inputs named "txt" and "nb".
`.updateBtn`	Should an update button be added to the controls ? If `TRUE`, then the graphic is updated only when the user clicks on the update button.
`.saveBtn`	Should an save button be added to the controls ? For saving output as html. Does not work in RStudio Viewer
`.exportBtn`	Should an export button be added to the controls ? For saving output as png. Does not work in RStudio Viewer

.exportType	.exportBtn, using html2canvas (default) and keeping current zoom, ... or using webshot
.viewer	Controls where the gadget should be displayed. "pane" corresponds to the Rstudio viewer, "window" to a dialog window, and "browser" to an external web browser.
.compare	Sometimes one wants to compare the same chart but with two different sets of parameters. This is the purpose of this argument. It can be a character vector of input names or a named list whose names are the names of the inputs that should vary between the two charts. Each element of the list must be a vector or a list of length equal to the number of charts with the initial values of the corresponding parameter for each chart. It can also be NULL. In this case, the parameter is initialized with the default value for the two charts.
.compareOpts	List of options created compareOptions. These options indicate the number of charts to create and their disposition.
.translations	List of translation strings created with function mwTranslations. Used to translate UI titles and labels.
.return	A function that can be used to modify the output of manipulateWidget. It must take two parameters: the first one is the final widget, the second one is a list of environments containing the input values of each individual widget. The length of this list is one if .compare is null, two or more if it has been defined.
.width	Width of the UI. Used only on Rmarkdown documents with option runtime: shiny.
.height	Height of the UI. Used only on Rmarkdown documents with option runtime: shiny.
.runApp	(advanced usage) If true, a shiny gadget is started. If false, the function returns a MWController object. This object can be used to check with command line instructions the behavior of the application. (See help page of MWController). Notice that this parameter is always false in a non-interactive session (for instance when running tests of a package).

Value

The result of the expression evaluated with the last values of the controls. It should be an object of class htmlWidget.

Advanced Usage

The "normal" use of the function is to provide an expression that always return an htmlwidget. In such case, every time the user changes the value of an input, the current widget is destroyed and a new one is created and rendered.

Some packages provide functions to update a widget that has already been rendered. This is the case for instance for package leaflet with the function leafletProxy. To use such functions, manipulateWidget evaluates the parameter .expr with four extra variables:

- .initial: TRUE if the expression is evaluated for the first time and then the widget has not been rendered yet, FALSE if the widget has already been rendered.

- `.session`: A shiny session object.
- `.output`: ID of the output in the shiny interface.
- `.id`: Id of the chart. It can be used in comparison mode to make further customization without the need to create additional input controls.

You can take a look at the last example to see how to use these two variables to update a leaflet widget.

Modify the returned widget

In some specific situations, a developer may want to use `manipulateWidget` in a function that waits the user to click on the "Done" button and modifies the widget returned by `manipulateWidget`. In such situation, parameter `.return` should be used so that `manipulateWidget` is the last function called. Indeed, if other code is present after, the custom function will act very weird in a Rmarkdown document with "runtime: shiny".

Examples

```
# Basic example with fake data
if (require(dygraphs)) {
  mydata <- data.frame(period = 1:100, value = rnorm(100))
  manipulateWidget(dygraph(mydata[range[1]:range[2], ], main = title),
                   range = mwSlider(1, 100, c(1, 100)),
                   title = mwText("Fictive time series"))
}

# Let use manipulateWidget to explore the evolution of energy consumption in
# the world
data("worldEnergyUse")

if (require(plotly)) {
  # Function that generates a chart representing the evolution of energy
  # consumption per country. Creating a function is not necessary. We do it
  # for clarity and reuse in the different examples.
  plotEnergyUse <- function(Country, Period, lwd = 2, col = "gray") {
    dataset <- subset(
      worldEnergyUse,
      country == Country & year >= Period[1] & year <= Period[2]
    )
    plot_ly(dataset) %>%
      add_lines(~year, ~energy_used, line = list(width = lwd, color = col)) %>%
      layout(title = paste("Energy used in", Country))
  }

  # Launch the interactive visualisation
  manipulateWidget(
    plotEnergyUse(Country, Period),
    Period = mwSlider(1960, 2014, c(1960, 2014)),
    Country = mwSelect(sort(unique(worldEnergyUse$country)), "United States")
  )
```

```
# Directly start comparison mode
manipulateWidget(
  plotEnergyUse(Country, Period),
  Period = mwSlider(1960, 2014, c(1960, 2014)),
  Country = mwSelect(sort(unique(worldEnergyUse$country))),
  .compare = list(Country = c("United States", "China")),
  .compareOpts = compareOptions(ncol = 2)
)

# Dynamic input parameters
#-------------------------
# The arguments of an input can depend on the values of other inputs.
# In this example, when the user changes the region, the choices of input
# "Country" are updated with the countries of that region.

# First we create a list that contains for each region the countries in that
# retion
refRegions <- by(worldEnergyUse$country, worldEnergyUse$region,
                 function(x) as.character(sort(unique(x))))

manipulateWidget(
  plotEnergyUse(Country, Period),
  Period = mwSlider(1960, 2014, c(1960, 2014)),
  Region = mwSelect(sort(unique(worldEnergyUse$region))),
  Country = mwSelect(choices = refRegions[[Region]])
)

# Grouping inputs
#----------------
# Inputs can be visually grouped with function mwGroup()
manipulateWidget(
  plotEnergyUse(Country, Period, lwd, col),
  Period = mwSlider(1960, 2014, c(1960, 2014)),
  Country = mwSelect(sort(unique(worldEnergyUse$country)), "United States"),
  `Graphical Parameters` = mwGroup(
    lwd = mwSlider(1,10, 2, label = "Line Width"),
    col = mwSelect(choices = c("gray", "black", "red"))
  )
)

# Conditional inputs
#-------------------
# Inputs can be displayed or hidden depending on the state of other inputs.
# In this example, user can choose to display the level of aggregation
# (region or country). Depending on the choixe, the application displays
# input Region or input Country.
plotEnergyUseRegion <- function(Region, Period, lwd = 2, col = "gray") {
  dataset <- subset(
    worldEnergyUse,
    region == Region & year >= Period[1] & year <= Period[2]
  )
  dataset <- aggregate(energy_used ~ year, sum, data = dataset)
```

```
    plot_ly(dataset) %>%
      add_lines(~year, ~energy_used, line = list(width = lwd, color = col)) %>%
      layout(title = paste("Energy used in", Region))
  }

  manipulateWidget(
    {
      if (Level == "Region") {
        plotEnergyUseRegion(Region, Period)
      } else {
        plotEnergyUse(Country, Period)
      }
    },
    Period = mwSlider(1960, 2014, c(1960, 2014)),
    Level = mwSelect(c("Region", "Country")),
    Region = mwSelect(sort(unique(worldEnergyUse$region)),
                      .display = Level == "Region"),
    Country = mwSelect(sort(unique(worldEnergyUse$country)),
                       .display = Level == "Country")
  )

}

# Advanced Usage
# --------------
# When .expr is evaluated with tehnical variables:
# .initial: is it the first evaluation?
# .outputId: integer representing the id of the chart
# .output: shiny output id
# .session: shiny session
# They can be used to update an already rendered widget instead of replacing
# it each time an input value is modified.
#
# In this example, we represent on a map, the energy use of countries.
# When the user changes an input, the map is not redrawn. Only the circle
# markers are updated.
if (require(leaflet)) {
  plotMap <- function(Year, MaxRadius = 30, .initial, .session, .output) {
    dataset <- subset(worldEnergyUse, year == Year)
    radius <- sqrt(dataset$energy_used) /
      max(sqrt(worldEnergyUse$energy_used), na.rm = TRUE) * MaxRadius

    if (.initial) { # map has not been rendered yet
      map <- leaflet() %>% addTiles()
    } else { # map already rendered
      map <- leafletProxy(.output, .session) %>% clearMarkers()
    }

    map %>% addCircleMarkers(dataset$long, dataset$lat, radius = radius,
                             color = "gray", weight = 0, fillOpacity = 0.7)
  }

  manipulateWidget(
```

```
    plotMap(Year, MaxRadius, .initial, .session, .output),
    Year = mwSlider(1960, 2014, 2014),
    MaxRadius = mwSlider(10, 50, 20)
  )
}
```

---

mwCheckbox *Add a checkbox to a manipulateWidget gadget*

Description

Add a checkbox to a manipulateWidget gadget

Usage

```
mwCheckbox(value = FALSE, label = NULL, ..., .display = TRUE)
```

Arguments

value	Initial value of the input.
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functioncheckboxInput
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if(require(plotly)) {
 manipulateWidget(
   {
       plot_ly(iris, x = ~Sepal.Length, y = ~Sepal.Width,
               color = ~Species, type = "scatter", mode = "markers") %>%
         layout(showlegend = legend)
   },
   legend = mwCheckbox(TRUE, "Show legend")
 )
}
```

mwCheckboxGroup *Add a group of checkboxes to a manipulateWidget gadget*

Description

Add a group of checkboxes to a manipulateWidget gadget

Usage

```
mwCheckboxGroup(choices, value = c(), label = NULL, ..., .display = TRUE)
```

Arguments

choices	Vector or list of choices. If it is named, then the names rather than the values are displayed to the user.
value	Vector containing the values initially selected
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functioncheckboxGroupInput
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  manipulateWidget(
    {
      if (length(species) == 0) mydata <- iris
      else mydata <- iris[iris$Species %in% species,]

      plot_ly(mydata, x = ~Sepal.Length, y = ~Sepal.Width,
              color = ~droplevels(Species), type = "scatter", mode = "markers")
    },
    species = mwCheckboxGroup(levels(iris$Species))
  )
}
```

`MWController-class` *Controller object of a manipulateWidget application*

Description

`MWController` is a reference class that is used to manage interaction with data and update of the view created by manipulateWidget. Only users who desire to create automatic tests for applications created with `manipulateWidget` should care about this object.

Fields

`ncharts` Number of charts in the application

`nrow` Number of rows.

`ncol` Number of columns.

`autoUpdate` Boolean indicating if charts should be automatically updated when a value changes. list with `value` and `initBtn` (not autoUpdate, if want first charts on init)

Methods

`getParams(name, chartId = 1)` Get parameters of an input for a given chart

`getValue(name, chartId = 1)` Get the value of a variable for a given chart.

`getValues(chartId = 1)` Get all values for a given chart.

`isVisible(name, chartId = 1)` Indicates if a given input is visible

`returnCharts()` Return all charts.

`setValue(name, value, chartId = 1, updateHTML = FALSE, reactive = FALSE)` Update the value of a variable for a given chart.

`setValueAll(name, value, updateHTML = TRUE)` Update the value of an input for all charts

`updateCharts()` Update all charts.

Testing a manipulateWidget application

When `manipulateWidget` is used in a test script, it returns a `MWController` object instead of starting a shiny gadget. This object has methods to modify inputs values and check the state of the application. This can be useful to automatically checks if your application behaves like desired. Here is some sample code that uses package `testthat`:

```
library("testthat")

controller <- manipulateWidget(
  x + y,
  x = mwSlider(0, 10, 5),
  y = mwSlider(0, x, 0),
  .compare = "y"
)
```

```
test_that("Two charts are created", {
  expect_equal(controller$ncharts, 2)
})

test_that("Parameter 'max' of 'y' is updated when 'x' changes", {
  expect_equal(controller$getParams("y", 1)$max, 5)
  expect_equal(controller$getParams("y", 2)$max, 5)
  controller$setValue("x", 3)
  expect_equal(controller$getParams("y", 1)$max, 3)
  expect_equal(controller$getParams("y", 2)$max, 3)
})
```

`mwDate` *Add a date picker to a manipulateWidget gadget*

Description

Add a date picker to a manipulateWidget gadget

Usage

```
mwDate(value = NULL, label = NULL, ..., .display = TRUE)
```

Arguments

`value`	Default value of the input.
`label`	Display label for the control. If `NULL`, the name of the corresponding variable is used.
`...`	Other arguments passed to function`dateInput`
`.display`	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: `mwCheckboxGroup()`, `mwCheckbox()`, `mwDateRange()`, `mwGroup()`, `mwNumeric()`, `mwPassword()`, `mwRadio()`, `mwSelectize()`, `mwSelect()`, `mwSharedValue()`, `mwSlider()`, `mwText()`

Examples

```
if (require(dygraphs) && require(xts)) {
  mydata <- xts(rnorm(365), order.by = as.Date("2017-01-01") + 0:364)

  manipulateWidget(
    dygraph(mydata) %>% dyEvent(date, "Your birthday"),
    date = mwDate("2017-03-27", label = "Your birthday date",
                  min = "2017-01-01", max = "2017-12-31")
  )
}
```

`mwDateRange` *Add a date range picker to a manipulateWidget gadget*

Description

Add a date range picker to a manipulateWidget gadget

Usage

```
mwDateRange(
  value = c(Sys.Date(), Sys.Date() + 1),
  label = NULL,
  ...,
  .display = TRUE
)
```

Arguments

`value`	Vector containing two dates (either Date objects pr a string in yyy-mm-dd format) representing the initial date range selected.
`label`	Display label for the control. If `NULL`, the name of the corresponding variable is used.
`...`	Other arguments passed to function`dateRangeInput`
`.display`	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

An Input object

See Also

Other controls: `mwCheckboxGroup()`, `mwCheckbox()`, `mwDate()`, `mwGroup()`, `mwNumeric()`, `mwPassword()`, `mwRadio()`, `mwSelectize()`, `mwSelect()`, `mwSharedValue()`, `mwSlider()`, `mwText()`

Examples

```
if (require(dygraphs) && require(xts)) {
  mydata <- xts(rnorm(365), order.by = as.Date("2017-01-01") + 0:364)

  manipulateWidget(
    dygraph(mydata) %>% dyShading(from=period[1], to = period[2], color = "#CCEBD6"),
    period = mwDateRange(c("2017-03-01", "2017-04-01"),
                  min = "2017-01-01", max = "2017-12-31")
  )
}
```

`mwGroup` *Group inputs in a collapsible box*

Description

This function generates a collapsible box containing inputs. It can be useful when there are a lot of inputs and one wants to group them.

Usage

```
mwGroup(..., label = NULL, .display = TRUE)
```

Arguments

`...`	inputs that will be grouped in the box
`label`	label of the group inputs
`.display`	expression that evaluates to TRUE or FALSE, indicating when the group should be shown/hidden.

Value

Input of type "group".

See Also

Other controls: `mwCheckboxGroup()`, `mwCheckbox()`, `mwDateRange()`, `mwDate()`, `mwNumeric()`, `mwPassword()`, `mwRadio()`, `mwSelectize()`, `mwSelect()`, `mwSharedValue()`, `mwSlider()`, `mwText()`

Examples

```
if(require(dygraphs)) {
  mydata <- data.frame(x = 1:100, y = rnorm(100))
  manipulateWidget(
    dygraph(mydata[range[1]:range[2], ],
            main = title, xlab = xlab, ylab = ylab),
    range = mwSlider(1, 100, c(1, 100)),
```

```
    "Graphical parameters" = mwGroup(
      title = mwText("Fictive time series"),
      xlab = mwText("X axis label"),
      ylab = mwText("Y axis label")
    )
  )
}
```

`mwModule` *Add a manipulateWidget to a shiny application*

Description

These two functions can be used to include a manipulateWidget object in a shiny application. `mwModuleUI` must be used in the UI to generate the required HTML elements and add javascript and css dependencies. `mwModule` must be called once in the server function of the application.

Usage

```
mwModule(id, controller, fillPage = FALSE, ...)

mwModuleUI(
  id,
  border = TRUE,
  okBtn = FALSE,
  saveBtn = TRUE,
  exportBtn = TRUE,
  updateBtn = FALSE,
  allowCompare = TRUE,
  margin = 0,
  width = "100%",
  height = 400,
  header = NULL,
  footer = NULL
)
```

Arguments

`id`	A unique string that identifies the module
`controller`	Object of class `MWController` returned by `manipulateWidget` when parameter `.runApp` is `FALSE`.
`fillPage`	: `logical`. Render in a fillPage or not ? Defaut to FALSE
`...`	named arguments containing reactive values. They can be used to send data from the main shiny application to the module.
`border`	Should a border be added to the module ?

okBtn	Should the UI contain the OK button ?
saveBtn	Should the UI contain the save button ? For saving output as html
exportBtn	Should an export button be added to the controls ? For saving output as png
updateBtn	Should the updateBtn be added to the UI ?
allowCompare	If TRUE (the default), then the user can use the UI to add or remove charts and choose which variables to compare
margin	Margin to apply around the module UI. Should be one two or four valid css units.
width	Width of the module UI.
height	Height of the module UI.
header	Tag or list of tags to display as a common header above all tabPanels.
footer	Tag or list of tags to display as a common footer below all tabPanels

Value

mwModuleUI returns the required HTML elements for the module. mwModule is only used for its side effects.

Examples

```
if (interactive() & require("dygraphs")) {
  require("shiny")
  ui <- fillPage(
  fillRow(
    flex = c(NA, 1),
    div(
      textInput("title", label = "Title", value = "glop"),
      selectInput("series", "series", choices = c("series1", "series2", "series3"))
    ),
    mwModuleUI("ui", height = "100%")
  ))

  server <- function(input, output, session) {
    mydata <- data.frame(
      year = 2000+1:100,
      series1 = rnorm(100),
      series2 = rnorm(100),
      series3 = rnorm(100)
    )

    c <- manipulateWidget(
      {
        dygraph(mydata[range[1]:range[2] - 2000, c("year", series)], main = title)
      },
      range = mwSlider(2001, 2100, c(2001, 2050)),
      series = mwSharedValue(),
      title = mwSharedValue(), .runApp = FALSE,
      .compare = "range"
    )
```

```
  #
  mwModule("ui", c, title = reactive(input$title), series = reactive(input$series))
}

shinyApp(ui, server)


}
```

---

mwNumeric *Add a numeric input to a manipulateWidget gadget*

Description

Add a numeric input to a manipulateWidget gadget

Usage

```
mwNumeric(value, label = NULL, ..., .display = TRUE)
```

Arguments

value	Initial value of the numeric input.
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functionnumericInput
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  manipulateWidget({
      plot_ly(data.frame(x = 1:10, y = rnorm(10, mean, sd)), x=~x, y=~y,
              type = "scatter", mode = "markers")
    },
    mean = mwNumeric(0),
    sd = mwNumeric(1, min = 0, step = 0.1)
```

```
  )
}
```

`mwPassword` *Add a password to a manipulateWidget gadget*

Description

Add a password to a manipulateWidget gadget

Usage

```
mwPassword(value = "", label = NULL, ..., .display = TRUE)
```

Arguments

`value`	Default value of the input.
`label`	Display label for the control. If `NULL`, the name of the corresponding variable is used.
`...`	Other arguments passed to function`passwordInput`
`.display`	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: `mwCheckboxGroup()`, `mwCheckbox()`, `mwDateRange()`, `mwDate()`, `mwGroup()`, `mwNumeric()`, `mwRadio()`, `mwSelectize()`, `mwSelect()`, `mwSharedValue()`, `mwSlider()`, `mwText()`

Examples

```
if (require(plotly)) {
  manipulateWidget(
    {
      if (passwd != 'abc123') {
        plot_ly(type = "scatter", mode="markers") %>%
          layout(title = "Wrong password. True password is 'abc123'")
      } else {
     plot_ly(data.frame(x = 1:10, y = rnorm(10)), x=~x, y=~y, type = "scatter", mode = "markers")
      }
    },
    user = mwText(label = "Username"),
    passwd = mwPassword(label = "Password")
  )
}
```

mwRadio *Add radio buttons to a manipulateWidget gadget*

Description

Add radio buttons to a manipulateWidget gadget

Usage

```
mwRadio(choices, value = NULL, label = NULL, ..., .display = TRUE)
```

Arguments

choices	Vector or list of choices. If it is named, then the names rather than the values are displayed to the user.
value	Initial value of the input. If not specified, the first choice is used.
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functionradioButtons
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwSelectize(), mwSelect(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  mydata <- data.frame(x = 1:100, y = rnorm(100))

  manipulateWidget(
    {
      mode <- switch(type, points = "markers", lines = "lines", both = "markers+lines")
      plot_ly(mydata, x=~x, y=~y, type = "scatter", mode = mode)
    },
    type = mwRadio(c("points", "lines", "both"))
  )
}
```

mwSelect *Add a Select list input to a manipulateWidget gadget*

Description

Add a Select list input to a manipulateWidget gadget

Usage

```
mwSelect(
  choices = value,
  value = NULL,
  label = NULL,
  ...,
  multiple = FALSE,
  .display = TRUE
)
```

Arguments

choices	Vector or list of choices. If it is named, then the names rather than the values are displayed to the user.
value	Initial value of the input. If not specified, the first choice is used.
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functionselectInput.
multiple	Is selection of multiple items allowed?
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  mydata <- data.frame(x = 1:100, y = rnorm(100))

  manipulateWidget(
    {
```

```
    mode <- switch(type, points = "markers", lines = "lines", both = "markers+lines")
    plot_ly(mydata, x=~x, y=~y, type = "scatter", mode = mode)
  },
  type = mwSelect(c("points", "lines", "both"))
)

Sys.sleep(0.5)

# Select multiple values
manipulateWidget(
  {
    if (length(species) == 0) mydata <- iris
    else mydata <- iris[iris$Species %in% species,]

    plot_ly(mydata, x = ~Sepal.Length, y = ~Sepal.Width,
            color = ~droplevels(Species), type = "scatter", mode = "markers")
  },
  species = mwSelect(levels(iris$Species), multiple = TRUE)
)
}
```

`mwSelectize` *Add a Select list input to a manipulateWidget gadget*

Description

Add a Select list input to a manipulateWidget gadget

Usage

```
mwSelectize(
  choices = value,
  value = NULL,
  label = NULL,
  ...,
  multiple = FALSE,
  options = NULL,
  .display = TRUE
)
```

Arguments

`choices`	Vector or list of choices. If it is named, then the names rather than the values are displayed to the user.
`value`	Initial value of the input. If not specified, the first choice is used.
`label`	Display label for the control. If `NULL`, the name of the corresponding variable is used.

...	Other arguments passed to functionselectInput.
multiple	Is selection of multiple items allowed?
options	A list of options. See the documentation of selectize.js for possible options
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelect(), mwSharedValue(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  mydata <- data.frame(x = 1:100, y = rnorm(100))

  # Select multiple values
  manipulateWidget(
    {
      if (length(species) == 0) mydata <- iris
      else mydata <- iris[iris$Species %in% species,]

      plot_ly(mydata, x = ~Sepal.Length, y = ~Sepal.Width,
              color = ~droplevels(Species), type = "scatter", mode = "markers")
    },
    species = mwSelectize(c("Select one or two species : " = "", levels(iris$Species)),
        multiple = TRUE, options = list(maxItems = 2))
  )
}
```

---

mwSharedValue *Shared Value*

Description

This function creates a virtual input that can be used to store a dynamic shared variable that is accessible in inputs as well as in output.

Usage

```
mwSharedValue(expr = NULL)
```

Arguments

expr	Expression used to compute the value of the input.

Value

An Input object of type "sharedValue".

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSlider(), mwText()

Examples

```
if (require(plotly)) {
  # Plot the characteristics of a car and compare with the average values for
  # cars with same number of cylinders.
  # The shared variable 'subsetCars' is used to avoid subsetting multiple times
  # the data: this value is updated only when input 'cylinders' changes.
  colMax <- apply(mtcars, 2, max)

  plotCar <- function(cardata, carName) {
    carValues <- unlist(cardata[carName, ])
    carValuesRel <- carValues / colMax

    avgValues <- round(colMeans(cardata), 2)
    avgValuesRel <- avgValues / colMax

    plot_ly() %>%
      add_bars(x = names(cardata), y = carValuesRel, text = carValues,
               hoverinfo = c("x+text"), name = carName) %>%
      add_bars(x = names(cardata), y = avgValuesRel, text = avgValues,
               hoverinfo = c("x+text"), name = "average") %>%
      layout(barmode = 'group')
  }

  c <- manipulateWidget(
    plotCar(subsetCars, car),
    cylinders = mwSelect(c("4", "6", "8")),
    subsetCars = mwSharedValue(subset(mtcars, cylinders == cyl)),
    car = mwSelect(choices = row.names(subsetCars))
  )
}
```

---

mwSlider *Add a Slider to a manipulateWidget gadget*

Description

Add a Slider to a manipulateWidget gadget

Usage

```
mwSlider(min, max, value, label = NULL, ..., .display = TRUE)
```

Arguments

min	The minimum value that can be selected.
max	The maximum value that can be selected.
value	Initial value of the slider A numeric vector of length one will create a regular slider; a numeric vector of length two will create a double-ended range slider
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functionsliderInput
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSharedValue(), mwText()

Examples

```
if (require(plotly)) {

  myWidget <- manipulateWidget(
   plot_ly(data.frame(x = 1:n, y = rnorm(n)), x=~x, y=~y, type = "scatter", mode = "markers"),
    n = mwSlider(1, 100, 10, label = "Number of values")
  )

  Sys.sleep(0.5)

  # Create a double ended slider to choose a range instead of a single value
  mydata <- data.frame(x = 1:100, y = rnorm(100))

  manipulateWidget(
    plot_ly(mydata[n[1]:n[2], ], x=~x, y=~y, type = "scatter", mode = "markers"),
    n = mwSlider(1, 100, c(1, 10), label = "Number of values")
  )

}
```

mwText *Add a text input to a manipulateWidget gadget*

Description

Add a text input to a manipulateWidget gadget

Usage

```
mwText(value = "", label = NULL, ..., .display = TRUE)
```

Arguments

value	Initial value of the text input.
label	Display label for the control. If NULL, the name of the corresponding variable is used.
...	Other arguments passed to functiontextInput
.display	expression that evaluates to TRUE or FALSE, indicating when the input control should be shown/hidden.

Value

A function that will generate the input control.

See Also

Other controls: mwCheckboxGroup(), mwCheckbox(), mwDateRange(), mwDate(), mwGroup(), mwNumeric(), mwPassword(), mwRadio(), mwSelectize(), mwSelect(), mwSharedValue(), mwSlider()

Examples

```
if (require(plotly)) {
  mydata <- data.frame(x = 1:100, y = rnorm(100))
  manipulateWidget({
      plot_ly(mydata, x=~x, y=~y, type = "scatter", mode = "markers") %>%
        layout(title = mytitle)
    },
    mytitle = mwText("Awesome title !")
  )
}
```

`mwTranslations` *Translate UI titles and labels*

Description

Creates a list of translation strings that can be passed to function `manipulateWidget` to translate some UI elements.

Usage

```
mwTranslations(
  settings = "Settings",
  chart = "Chart",
  compare = "Compare",
  compareVars = "Variables to compare",
  ncol = "Nb Columns",
  ncharts = "Nb Charts"
)
```

Arguments

`settings`	Title of the settings panel.
`chart`	Title of the chart panel.
`compare`	Label of the checkbox that activate the comparison mode.
`compareVars`	Label of the input containing the list of variables to compare.
`ncol`	Label of the input that sets the number of columns.
`ncharts`	Label of the input that sets the number of charts.

Value

Named list of translation strings.

Examples

```
translations <- mwTranslations(
  settings = "Parametres", chart = "Graphique", compare = "Comparaison",
  compareVars = "Variable de comparaison", ncharts = "Nb graph.", ncol = "Nb col."
)

if (require(dygraphs)) {
  mydata <- data.frame(year = 2000+1:100, value = rnorm(100))
  manipulateWidget(dygraph(mydata[range[1]:range[2] - 2000, ], main = title),
                   range = mwSlider(2001, 2100, c(2001, 2100)),
                   title = mwText("Fictive time series"),
                   .translations = translations)
}
```

staticPlot *Include a static image in a combinedWidgets*

Description

staticPlot is a function that generates a static plot and then return the HTML code needed to include the plot in a combinedWidgets. staticImage is a more general function that generates the HTML code necessary to include any image file.

Usage

```
staticPlot(expr, width = 600, height = 400)

staticImage(file, style = "max-width:100%;max-height:100%")
```

Arguments

expr	Expression that creates a static plot.
width	Width of the image to create.
height	Height of the image to create.
file	path of the image to include.
style	CSS style to apply to the image.

Value

a shiny.tag object containing the HTML code required to include the image or the plot in a combinedWidgets object.

Examples

```
staticPlot(hist(rnorm(100)))

if (require(plotly)) {
  data(iris)

  combineWidgets(
    plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
    staticPlot(hist(iris$Sepal.Length, breaks = 20), height = 300)
  )

  # You can also embed static images in the header, footer, left or right
  # columns of a combinedWidgets. The advantage is that the space allocated
  # to the static plot will be constant when the window is resized.

  combineWidgets(
    plot_ly(iris, x = ~Sepal.Length, type = "histogram", nbinsx = 20),
    footer = staticPlot(hist(iris$Sepal.Length, breaks = 20), height = 300)
  )
```

```
}
```

`summary.MWController` *summary method for MWController object*

Description

summary method for MWController object

Usage

```
## S3 method for class 'MWController'
summary(object, ...)
```

Arguments

`object`	MWController object
`...`	Not use

`worldEnergyUse` *Evolution of energy use per country*

Description

Data.frame containing energy consumption per country from 1960 to 2014. The data comes from the World Bank website. It contains one line per couple(country, year) and has the following columns:

Usage

```
worldEnergyUse
```

Format

An object of class `data.frame` with 9375 rows and 15 columns.

Details

- country Country name
- iso2c Country code in two characters
- year Year
- population Population of the country
- energy_used_per_capita Energy used per capita in kg of oil equivalent (EG.USE.PCAP.KG.OE)
- energy_imported_prop Proportion of energy used that has been imported (EG.IMP.CONS.ZS)
- energy_fossil_prop Fossil fuel energy consumption in proportion of total consumption (EG.USE.COMM.FO.ZS)
- energy_used Energy consumption in kg of oil equivalent
- energy_fossil Fossil fuel energy consumption in kg of oil equivalent
- prop_world_energy_used Share of the country in the world energy consumption
- prop_world_energy_fossil Share of the country in the world fossil energy consumption
- prop_world_population Share of the country in the world population
- long Longitude of the country
- lat Lattitude of the country
- region Region of the country

Author(s)

François Guillem <guillem.francois@gmail.com>

References

https://data.worldbank.org/indicator

Index