Package ‘DT’

January 20, 2025

Type Package

Title A Wrapper of the JavaScript Library 'DataTables'

Version 0.33

Description Data objects in R can be rendered as HTML tables using the JavaScript library 'DataTables' (typically via R Markdown or Shiny). The 'DataTables' library has been included in this R package. The package name 'DT' is an abbreviation of 'DataTables'.

URL https://github.com/rstudio/DT

BugReports https://github.com/rstudio/DT/issues

License GPL-3 | file LICENSE

Imports htmltools (>= 0.3.6), htmlwidgets (>= 1.3), httpuv, jsonlite (>= 0.9.16), magrittr, crosstalk, jquerylib, promises

Suggests knitr (>= 1.8), rmarkdown, shiny (>= 1.6), bslib, future, testit, tibble

VignetteBuilder knitr

RoxygenNote 7.3.1

Encoding UTF-8

NeedsCompilation no

Author Yihui Xie [aut],
Joe Cheng [aut, cre],
Xianying Tan [aut],
JJ Allaire [ctb],
Maximilian Girlich [ctb],
Greg Freedman Ellis [ctb],
Johannes Rauh [ctb],
SpryMedia Limited [ctb, cph] (DataTables in htmlwidgets/lib),
Brian Reavis [ctb, cph] (selectize.js in htmlwidgets/lib),
Leon Gersen [ctb, cph] (noUiSlider in htmlwidgets/lib),
Bartek Szopka [ctb, cph] (jquery.highlight.js in htmlwidgets/lib),
Alex Pickering [ctb],
William Holmes [ctb],

Mikko Marttila [ctb],
Andres Quintero [ctb],
Stéphane Laurent [ctb],
Posit Software, PBC [cph, fnd]

Maintainer Joe Cheng <`joe@posit.co`>

Repository CRAN

Date/Publication 2024-04-04 05:03:17 UTC

Contents

`coerceValue` *Coerce a character string to the same type as a target value*

Description

Create a new value from a character string based on an old value, e.g., if the old value is an integer, call `as.integer()` to coerce the string to an integer.

Usage

```
coerceValue(val, old)
```

Arguments

`val`	A character string.
`old`	An old value, whose type is the target type of `val`.

Details

This function only works with integer, double, date, time (`POSIXlt` or `POSIXct`), and factor values. The date must be of the format `%Y-%m-%dT%H:%M:%SZ`. The factor value must be in the levels of `old`, otherwise it will be coerced to `NA`.

Value

A value of the same data type as `old` if possible.

Examples

```
library(DT)
coerceValue("100", 1L)
coerceValue("1.23", 3.1416)
coerceValue("2018-02-14", Sys.Date())
coerceValue("2018-02-14T22:18:52Z", Sys.time())
coerceValue("setosa", iris$Species)
coerceValue("setosa2", iris$Species)  # NA
coerceValue("FALSE", TRUE)  # not supported
```

`datatable` *Create an HTML table widget using the DataTables library*

Description

This function creates an HTML widget to display rectangular data (a matrix or data frame) using the JavaScript library DataTables.

Usage

```
datatable(
  data,
  options = list(),
  class = "display",
  callback = JS("return table;"),
  rownames,
  colnames,
  container,
  caption = NULL,
  filter = c("none", "bottom", "top"),
  escape = TRUE,
  style = "auto",
  width = NULL,
  height = NULL,
  elementId = NULL,
  fillContainer = getOption("DT.fillContainer", NULL),
  autoHideNavigation = getOption("DT.autoHideNavigation", NULL),
  selection = c("multiple", "single", "none"),
  extensions = list(),
  plugins = NULL,
  editable = FALSE
)
```

Arguments

`data`	a data object (either a matrix or a data frame)
`options`	a list of initialization options (see `https://datatables.net/reference/option/`); the character options wrapped in `JS()` will be treated as literal JavaScript code instead of normal character strings; you can also set options globally via `options(DT.options = list(...))`, and global options will be merged into this `options` argument if set
`class`	the CSS class(es) of the table; see `https://datatables.net/manual/styling/classes`
`callback`	the body of a JavaScript callback function with the argument `table` to be applied to the DataTables instance (i.e. `table`)
`rownames`	`TRUE` (show row names) or `FALSE` (hide row names) or a character vector of row names; by default, the row names are displayed in the first column of the table if exist (not `NULL`)
`colnames`	if missing, the column names of the data; otherwise it can be an unnamed character vector of names you want to show in the table header instead of the default data column names; alternatively, you can provide a *named* numeric or character vector of the form `'newName1' = i1, 'newName2' = i2` or `c('newName1' = 'oldName1', 'newName2' = 'oldName2', ...)`, where `newName` is the new name you want to show in the table, and `i` or `oldName` is the index of the current column name
`container`	a sketch of the HTML table to be filled with data cells; by default, it is generated from `htmltools::tags$table()` with a table header consisting of the column names of the data
`caption`	the table caption; a character vector or a tag object generated from `htmltools::tags$caption()`
`filter`	whether/where to use column filters; `none`: no filters; `bottom/top`: put column filters at the bottom/top of the table; range sliders are used to filter numeric/date/time columns, select lists are used for factor columns, and text input boxes are used for character columns; if you want more control over the styles of filters, you can provide a named list to this argument; see Details for more
`escape`	whether to escape HTML entities in the table: `TRUE` means to escape the whole table, and `FALSE` means not to escape it; alternatively, you can specify numeric column indices or column names to indicate which columns to escape, e.g. `1:5` (the first 5 columns), `c(1, 3, 4)`, or `c(-1, -3)` (all columns except the first and third), or `c('Species', 'Sepal.Length')`; since the row names take the first column to display, you should add the numeric column indices by one when using `rownames`
`style`	either `'auto'`, `'default'`, `'bootstrap'`, or `'bootstrap4'`. If `'auto'`, and a **bslib** theme is currently active, then bootstrap styling is used in a way that "just works" for the active theme. Otherwise, DataTables `'default'` styling is used. If set explicitly to `'bootstrap'` or `'bootstrap4'`, one must take care to ensure Bootstrap's HTML dependencies (as well as Bootswatch themes, if desired) are included on the page. Note, when set explicitly, it's the user's responsibility to ensure that only one unique 'style' value is used on the same page, if multiple DT tables exist, as different styling resources may conflict with each other.

`width, height`	Width/Height in pixels (optional, defaults to automatic sizing)
`elementId`	An id for the widget (a random string by default).
`fillContainer`	`TRUE` to configure the table to automatically fill it's containing element. If the table can't fit fully into it's container then vertical and/or horizontal scrolling of the table cells will occur.
`autoHideNavigation`	`TRUE` to automatically hide navigational UI (only display the table body) when the number of total records is less than the page size. Note, it only works on the client-side processing mode and the 'pageLength' option should be provided explicitly.
`selection`	the row/column selection mode (single or multiple selection or disable selection) when a table widget is rendered in a Shiny app; alternatively, you can use a list of the form `list(mode = 'multiple',selected = c(1, 3, 8), target = 'row', selectable = c(-2, -3))` to pre-select rows and control the selectable range; the element `target` in the list can be `'column'` to enable column selection, or `'row+column'` to make it possible to select both rows and columns (click on the footer to select columns), or `'cell'` to select cells. See details section for more info.
`extensions`	a character vector of the names of the DataTables extensions (`https://datatables.net/extensions/index`)
`plugins`	a character vector of the names of DataTables plug-ins (`https://rstudio.github.io/DT/plugins.html`). Note that only those plugins supported by the `DT` package can be used here. You can see the available plugins by calling `DT:::available_plugins()`
`editable`	`FALSE` to disable the table editor, or `TRUE` (or `"cell"`) to enable editing a single cell. Alternatively, you can set it to `"row"` to be able to edit a row, or `"column"` to edit a column, or `"all"` to edit all cells on the current page of the table. In all modes, start editing by doubleclicking on a cell. This argument can also be a list of the form `list(target = TARGET, disable = list(columns = INDICES))`, where `TARGET` can be `"cell"`, `"row"`, `"column"`, or `"all"`, and `INDICES` is an integer vector of column indices. Use the list form if you want to disable editing certain columns. You can also restrict the editing to accept only numbers by setting this argument to a list of the form `list(target = TARGET, numeric = INDICES)` where `INDICES` can be the vector of the indices of the columns for which you want to restrict the editing to numbers or `"all"` to restrict the editing to numbers for all columns. If you don't set `numeric`, then the editing is restricted to numbers for all numeric columns; set `numeric = "none"` to disable this behavior. It is also possible to edit the cells in text areas, which are useful for large contents. For that, set the `editable` argument to a list of the form `list(target = TARGET, area = INDICES)` where `INDICES` can be the vector of the indices of the columns for which you want the text areas, or `"all"` if you want the text areas for all columns. Of course, you can request the numeric editing for some columns and the text areas for some other columns by setting `editable` to a list of the form `list(target = TARGET, numeric = INDICES1, area = INDICES2)`. Finally, you can edit date cells with a calendar with `list(target = TARGET, date = INDICES)`; the target columns must have

the `Date` type. If you don't set `date` in the `editable` list, the editing with the calendar is automatically set for all `Date` columns.

Details

`selection`:

1. The argument could be a scalar string, which means the selection `mode`, whose value could be one of `'multiple'` (the default), `'single'` and `'none'` (disable selection).
2. When a list form is provided for this argument, only parts of the "full" list are allowed. The default values for non-matched elements are `list(mode = 'multiple', selected = NULL, target = 'row', selectable = NULL)`.
3. `target` must be one of `'row'`, `'column'`, `'row+column'` and `'cell'`.
4. `selected` could be `NULL` or "indices".
5. `selectable` could be `NULL`, `TRUE`, `FALSE` or "indices", where `NULL` and `TRUE` mean all the table is selectable. When `FALSE`, it means users can't select the table by the cursor (but they could still be able to select the table via `dataTableProxy`, specifying `ignore.selectable = TRUE`). If "indices", they must be all positive or non-positive values. All positive "indices" mean only the specified ranges are selectable while all non-positive "indices" mean those ranges are *not* selectable. The "indices"' format is specified below.
6. The "indices"' format of `selected` and `selectable`: when `target` is `'row'` or `'column'`, it should be a plain numeric vector; when `target` is `'row+column'`, it should be a list, specifying `rows` and `cols` respectively, e.g., `list(rows = 1, cols = 2)`; when `target` is `'cell'`, it should be a 2-col `matrix`, where the two values of each row stand for the row and column index.
7. Note that DT has its own selection implementation and doesn't use the Select extension because the latter doesn't support the server-side processing mode well. Please set this argument to `'none'` if you really want to use the Select extension.

`options$columnDefs`:

1. `columnDefs` is an option that provided by the DataTables library itself, where the user can set various attributes for columns. It must be provided as a list of list, where each sub-list must contain a vector named 'targets', specifying the applied columns, i.e., `list(list(..., targets = '_all'), list(..., targets = c(1, 2)))`
2. `columnDefs$targets` is a vector and should be one of:
   - 0 or a positive integer: column index counting from the left.
   - A negative integer: column index counting from the right.
   - A string: the column name. Note, it must be the names of the original data, not the ones that (could) be changed via param `colnames`.
   - The string "_all": all columns (i.e. assign a default).
3. When conflicts happen, e.g., a single column is defined for some property twice but with different values, the value that defined earlier takes the priority. For example, `list(list(visible=FALSE, target=1), list(visible=TRUE, target=1))` results in a table whose first column is *invisible*.
4. See `https://datatables.net/reference/option/columnDefs` for more.

`filter`:

1. `filter` can be used to position and customize column filters. A scalar string value defines the position, and must be one of `'none'` (the default), `'bottom'` and `'top'`. A named list can be used for further control. In the named list form:
2. `$position` is a string as described above. It defaults to `'none'`.
3. `$clear` is a logical value indicating if clear buttons should appear in input boxes. It defaults to `TRUE`.
4. `$plain` is a logical value indicating if plain styling should be used for input boxes instead of Bootstrap styling. It defaults to `FALSE`.
5. `$vertical` is a logical value indicating if slider widgets should be oriented vertically rather than horizontally. It defaults to `FALSE`.
6. `$opacity` is a numeric value between 0 and 1 used to set the level of transparency of slider widgets. It defaults to `1`.
7. `$settings` is a named list used to directly pass configuration for initializing filter widgets in JavaScript.
    - The `$select` element is passed to the select widget, and `$slider` is passed to the slider widget.
    - Valid values depend on the settings accepted by the underlying JavaScript libraries, Selectize and noUiSlider. Please note that the versions bundled with DT are currently quite old, so accepted settings may not match their most recent documentation.
    - These settings can override values set by DT, so specifying a setting already in use may break something. Use with care.

Note

You are recommended to escape the table content for security reasons (e.g. XSS attacks) when using this function in Shiny or any other dynamic web applications.

References

See `https://rstudio.github.io/DT/` for the full documentation.

Examples

```
library(DT)

# see the package vignette for examples and the link to website
vignette('DT', package = 'DT')

# some boring edge cases for testing purposes
m = matrix(nrow = 0, ncol = 5, dimnames = list(NULL, letters[1:5]))
datatable(m)  # zero rows
datatable(as.data.frame(m))

m = matrix(1, dimnames = list(NULL, 'a'))
datatable(m)  # one row and one column
datatable(as.data.frame(m))
```

```
m = data.frame(a = 1, b = 2, c = 3)
datatable(m)
datatable(as.matrix(m))

# dates
datatable(data.frame(
  date = seq(as.Date("2015-01-01"), by = "day", length.out = 5), x = 1:5
))
datatable(data.frame(x = Sys.Date()))
datatable(data.frame(x = Sys.time()))
```

`dataTableAjax` *Register a data object in a shiny session for DataTables*

Description

This function stores a data object in a shiny session and returns a URL that returns JSON data based on DataTables Ajax requests. The URL can be used as the `url` option inside the `ajax` option of the table. It is basically an implementation of server-side processing of DataTables in R. Filtering, sorting, and pagination are processed through R instead of JavaScript (client-side processing).

Usage

```
dataTableAjax(
  session,
  data,
  rownames,
  filter = dataTablesFilter,
  outputId,
  future = FALSE
)
```

Arguments

`session`	the `session` object in the shiny server function (`function(input, output, session)`)
`data`	a data object (will be coerced to a data frame internally)
`rownames`	see `datatable()`; it must be consistent with what you use in `datatable()`, e.g. if the widget is generated by `datatable(rownames = FALSE)`, you must also use `dataTableAjax(rownames = FALSE)` here
`filter`	(for expert use only) a function with two arguments `data` and `params` (Ajax parameters, a list of the form `list(search = list(value = 'FOO', regex = 'false'), length = 10, ...)`) that return the filtered table result according to the DataTables Ajax request
`outputId`	the output ID of the table (the same ID passed to `dataTableOutput()`; if missing, an attempt to infer it from `session` is made. If it can't be inferred, a random id is generated.)

`future` whether the server-side filter function should be executed as a future or as a standard synchronous function. If true, the future will be evaluated according to the session's plan.

Details

Normally you should not need to call this function directly. It is called internally when a table widget is rendered in a Shiny app to configure the table option `ajax` automatically. If you are familiar with **DataTables**' server-side processing, and want to use a custom filter function, you may call this function to get an Ajax URL.

Value

A character string (an Ajax URL that can be queried by DataTables).

References

`https://rstudio.github.io/DT/server.html`

Examples

```
DTApp = function(data, ..., options = list()) {
  library(shiny)
  library(DT)
  shinyApp(
    ui = fluidPage(
      title = 'Server-side processing of DataTables',
      fluidRow(
        DT::dataTableOutput('tbl')
      )
    ),
    server = function(input, output, session) {
      options$serverSide = TRUE
      options$ajax = list(url = dataTableAjax(session, data, outputId = 'tbl'))
      # create a widget using an Ajax URL created above
      widget = datatable(data, ..., options = options)
      output$tbl = DT::renderDataTable(widget)
    }
  )
}

if (interactive()) DTApp(iris)
if (interactive()) DTApp(iris, filter = 'top')
```

`dataTableOutput` *Helper functions for using DT in Shiny*

Description

These two functions are like most `fooOutput()` and `renderFoo()` functions in the **shiny** package. The former is used to create a container for table, and the latter is used in the server logic to render the table.

Usage

```
dataTableOutput(outputId, width = "100%", height = "auto", fill = TRUE)

DTOutput(outputId, width = "100%", height = "auto", fill = TRUE)

renderDataTable(
  expr,
  server = TRUE,
  env = parent.frame(),
  quoted = FALSE,
  funcFilter = dataTablesFilter,
  future = FALSE,
  outputArgs = list(),
  ...
)

renderDT(
  expr,
  server = TRUE,
  env = parent.frame(),
  quoted = FALSE,
  funcFilter = dataTablesFilter,
  future = FALSE,
  outputArgs = list(),
  ...
)
```

Arguments

`outputId`	output variable to read the table from
`width`	the width of the table container
`height`	the height of the table container
`fill`	passed to `htmlwidgets::shinyWidgetOutput()`, see there for explanation (requires **htmlwidgets** > v1.5.4).
`expr`	an expression to create a table widget (normally via `datatable()`), or a data object to be passed to `datatable()` to create a table widget
`server`	whether to use server-side processing. If `TRUE`, then the data is kept on the server and the browser requests a page at a time; if `FALSE`, then the entire data frame is sent to the browser at once. Highly recommended for medium to large data frames, which can cause browsers to slow down or crash. Note that if you want to use `renderDataTable` with `shiny::bindCache()`, this must be `FALSE`.

`env`	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If `expr` is a quosure and `quoted` is `TRUE`, then `env` is ignored.
`quoted`	If it is `TRUE`, then the `quote()`ed value of `expr` will be used when `expr` is evaluated. If `expr` is a quosure and you would like to use its expression as a value for `expr`, then you must set `quoted` to `TRUE`.
`funcFilter`	(for expert use only) passed to the `filter` argument of `dataTableAjax()`
`future`	whether the server-side filter function should be executed as a future or as a standard synchronous function. If true, the future will be evaluated according to the session's plan.
`outputArgs`	A list of arguments to be passed through to the implicit call to `dataTableOutput()` when `renderDataTable()` is used in an interactive R Markdown document.
`...`	ignored when `expr` returns a table widget, and passed as additional arguments to `datatable()` when `expr` returns a data object

References

`https://rstudio.github.io/DT/shiny.html`

Examples

```
if (interactive()) {
  library(shiny)
  library(DT)
  shinyApp(
    ui = fluidPage(fluidRow(column(12, DTOutput('tbl')))),
    server = function(input, output) {
      output$tbl = renderDT(
        iris, options = list(lengthChange = FALSE)
      )
    }
  )
}
```

`dataTableProxy` *Manipulate an existing DataTables instance in a Shiny app*

Description

The function `dataTableProxy()` creates a proxy object that can be used to manipulate an existing DataTables instance in a Shiny app, e.g. select rows/columns, or add rows.

Usage

```
dataTableProxy(
  outputId,
  session = shiny::getDefaultReactiveDomain(),
  deferUntilFlush = TRUE
)

selectRows(proxy, selected, ignore.selectable = FALSE)

selectColumns(proxy, selected, ignore.selectable = FALSE)

selectCells(proxy, selected, ignore.selectable = FALSE)

addRow(proxy, data, resetPaging = TRUE)

clearSearch(proxy)

selectPage(proxy, page)

updateCaption(proxy, caption)

updateSearch(proxy, keywords = list(global = NULL, columns = NULL))

showCols(proxy, show, reset = FALSE)

hideCols(proxy, hide, reset = FALSE)

colReorder(proxy, order, origOrder = FALSE)

reloadData(
  proxy,
  resetPaging = TRUE,
  clearSelection = c("all", "none", "row", "column", "cell")
)
```

Arguments

`outputId`	the id of the table to be manipulated (the same id as the one you used in `dataTableOutput()`)
`session`	the Shiny session object (from the server function of the Shiny app)
`deferUntilFlush`	whether an action should be carried out right away, or should be held until after the next time all of the outputs are updated
`proxy`	a proxy object returned by `dataTableProxy()`
`selected`	an integer vector of row/column indices, or a matrix of two columns (row and column indices, respectively) for cell indices; you may use `NULL` to clear existing selections

`ignore.selectable`	when `FALSE` (the default), the "non-selectable" range specified by `selection = list(selectable= )` is respected, i.e., you can't select "non-selectable" range. Otherwise, it is ignored.
`data`	a single row of data to be added to the table; it can be a matrix or data frame of one row, or a vector or list of row data (in the latter case, please be cautious about the row name: if your table contains row names, here `data` must also contain the row name as the first element)
`resetPaging`	whether to reset the paging position
`page`	a number indicating the page to select
`caption`	a new table caption (see the `caption` argument of `datatable()`)
`keywords`	a list of two components: `global` is the global search keyword of a single character string (ignored if `NULL`); `columns` is a character vector of the search keywords for all columns (when the table has one column for the row names, this vector of keywords should contain one keyword for the row names as well)
`show`	a vector of column positions to show (the indexing starts at 0, but if row.names are visible, they are the first column).
`reset`	if `TRUE`, will only show/hide the columns indicated.
`hide`	a vector of column positions to hide
`order`	A numeric vector of column positions, starting from 0, and including the row.names as a column, if they are include. Must contain a value for all columns, regardless of whether they are visible or not. Also for column reordering to work, the datatable must have extension 'ColReorder' set as well as option 'colReordoer' set to TRUE).
`origOrder`	Whether column reordering should be relative to the original order (the default is to compare to current order)
`clearSelection`	which existing selections to clear: it can be any combinations of `row`, `column`, and `cell`, or `all` for all three, or `none` to keep current selections (by default, all selections are cleared after the data is reloaded)

Note

`addRow()` only works for client-side tables. If you want to use it in a Shiny app, make sure to use `renderDataTable(..., server = FALSE)`. Also note that the column filters (if used) of the table will not be automatically updated when a new row is added, e.g., the range of the slider of a column will stay the same even if you have added a value outside the range of the original data column.

`reloadData()` only works for tables in the server-side processing mode, e.g. tables rendered with `renderDataTable(server = TRUE)`. The data to be reloaded (i.e. the one you pass to `dataTableAjax()`) must have exactly the same number of columns as the previous data object in the table.

References

https://rstudio.github.io/DT/shiny.html

`doColumnSearch` *Server-side searching*

Description

`doGlobalSearch()` can be used to search a data frame given the search string typed by the user into the global search box of a `datatable`. `doColumnSearch()` does the same for a vector given the search string typed into a column filter. These functions are used internally by the default `filter` function passed to `dataTableAjax()`, allowing you to replicate the search results that server-side processing returns.

Usage

```
doColumnSearch(x, search_string, options = list())

doGlobalSearch(data, search_string, options = list())
```

Arguments

`x`	a vector, the type of which determines the expected `search_string` format
`search_string`	a string that determines what to search for. The format depends on the type of input, matching what a user would type in the associated filter control.
`options`	a list of options used to control how searching character values works. Supported options are `regex`, `caseInsensitive` and (for global search) `smart`.
`data`	a data frame

Value

An integer vector of filtered row indices

See Also

The column filters section online for search string formats: `https://rstudio.github.io/DT/`

Accessing the search strings typed by a user in a Shiny app: `https://rstudio.github.io/DT/shiny.html`

Examples

```
doGlobalSearch(iris, "versi")
doGlobalSearch(iris, "v.r.i", options = list(regex = TRUE))

doColumnSearch(iris$Species, "[\"versicolor\"]")
doColumnSearch(iris$Sepal.Length, "4 ... 5")
```

`DT-imports` *Objects imported from other packages*

Description

These objects are imported from other packages. Follow the links to their documentation.

htmlwidgets `JS`, `saveWidget`

magrittr `%>%`

`editData` *Edit a data object using the information from the editor in a DataTable*

Description

When editing cells in a DataTable in a Shiny app, we know the row/column indices and values of the cells that were edited. With these information, we can update the data object behind the DataTable accordingly.

Usage

```
editData(data, info, proxy = NULL, rownames = TRUE, resetPaging = FALSE, ...)
```

Arguments

`data`	The original data object used in the DataTable.
`info`	The information about the edited cells. It should be obtained from `input$tableId_cell_edit` from Shiny, and is a data frame containing columns `row`, `col`, and `value`.
`proxy`, `resetPaging`, `...`	(Optional) If `proxy` is provided, it must be either a character string of the output ID of the table or a proxy object created from `dataTableProxy()`, and the rest of arguments are passed to `replaceData()` to update the data in a DataTable instance in a Shiny app.
`rownames`	Whether row names are displayed in the table.

Value

The updated data object.

Note

For factor columns, new levels would be automatically added when necessary to avoid `NA` coercing.

`formatCurrency` *Format table columns*

Description

Format numeric columns in a table as currency (`formatCurrency()`) or percentages (`formatPercentage()`), or round numbers to a specified number of decimal places (`formatRound()`), or a specified number of significant figures (`formatSignif()`). The function `formatStyle()` applies CSS styles to table cells by column.

Usage

```
formatCurrency(
  table,
  columns,
  currency = "$",
  interval = 3,
  mark = ",",
  digits = 2,
  dec.mark = getOption("OutDec"),
  before = TRUE,
  zero.print = NULL,
  rows = NULL
)

formatString(table, columns, prefix = "", suffix = "", rows = NULL)

formatPercentage(
  table,
  columns,
  digits = 0,
  interval = 3,
  mark = ",",
  dec.mark = getOption("OutDec"),
  zero.print = NULL,
  rows = NULL
)

formatRound(
  table,
  columns,
  digits = 2,
  interval = 3,
  mark = ",",
  dec.mark = getOption("OutDec"),
  zero.print = NULL,
  rows = NULL
)

formatSignif(
  table,
  columns,
  digits = 2,
  interval = 3,
  mark = ",",
  dec.mark = getOption("OutDec"),
  zero.print = NULL,
  rows = NULL
)

formatDate(table, columns, method = "toDateString", params = NULL, rows = NULL)

formatStyle(
  table,
  columns,
  valueColumns = columns,
  target = c("cell", "row"),
  fontWeight = NULL,
  color = NULL,
  backgroundColor = NULL,
  background = NULL,
  ...
)
```

Arguments

`table`	a table object created from `datatable()`
`columns`	the indices of the columns to be formatted (can be character, numeric, logical, or a formula of the form `~ V1 + V2`, which is equivalent to `c('V1', 'V2')`)
`currency`	the currency symbol
`interval`	put a marker after how many digits of the numbers
`mark`	the marker after every `interval` decimals in the numbers
`digits`	the number of decimal places to round to
`dec.mark`	a character to indicate the decimal point
`before`	whether to place the currency symbol before or after the values
`zero.print`	a string to specify how zeros should be formatted. Useful for when many zero values exist. If `NULL`, keeps zero as it is.
`rows`	an integer vector (starting from 1) to specify the only rows that the style applies to. By default, it's `NULL`, meaning all rows should be formatted. Note, `formatStyle()` doesn't support this argument and you should use `styleRow()` instead. In addition, this only works expected in the client-side processing mode, i.e., `server = FALSE`.
`prefix`	string to put in front of the column values

`suffix`	string to put after the column values
`method`	the method(s) to convert a date to string in JavaScript; see `DT:::DateMethods` for a list of possible methods, and `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date` for a full reference
`params`	a list parameters for the specific date conversion method, e.g., for the `toLocaleDateString()` method, your browser may support `params = list('ko-KR', list(year = 'numeric', month = 'long', day = 'numeric'))`
`valueColumns`	indices of the columns from which the cell values are obtained; this can be different with the `columns` argument, e.g. you may style one column based on the values of a different column
`target`	the target to apply the CSS styles to (the current cell or the full row)
`fontWeight`	the font weight, e.g. `'bold'` and `'normal'`
`color`	the font color, e.g. `'red'` and `'#ee00aa'`
`backgroundColor`	the background color of table cells
`background`	the background of table cells
`...`	other CSS properties, e.g. `'border'`, `'font-size'`, `'text-align'`, and so on; if you want to condition CSS styles on the cell values, you may use the helper functions such as `styleInterval()`; note the actual CSS property names are dash-separated, but you can use camelCase names in this function (otherwise you will have to use backticks to quote the names, e.g. `` `font-size` = '12px' ``), and this function will automatically convert camelCase names to dash-separated names (e.g. `'fontWeight'` will be converted to `'font-weight'` internally)

Note

The length of arguments other than `table` should be 1 or the same as the length of `columns`.

References

See `https://rstudio.github.io/DT/functions.html` for detailed documentation and examples.

Examples

```
library(DT)
m = cbind(matrix(rnorm(120, 1e5, 1e6), 40), runif(40), rnorm(40, 100))
colnames(m) = head(LETTERS, ncol(m))
m

# format the columns A and C as currency, and D as percentages
datatable(m) %>% formatCurrency(c('A', 'C')) %>% formatPercentage('D', 2)

# the first two columns are Euro currency, and round column E to 3 decimal places
datatable(m) %>% formatCurrency(1:2, '\U20AC') %>% formatRound('E', 3)

# render vapor pressure with only two significant figures.
datatable(pressure) %>% formatSignif('pressure',2)
```

```
# apply CSS styles to columns
datatable(iris) %>%
  formatStyle('Sepal.Length', fontWeight = styleInterval(5, c('bold', 'weight'))) %>%
  formatStyle('Sepal.Width',
    color = styleInterval(3.4, c('red', 'white')),
    backgroundColor = styleInterval(3.4, c('yellow', 'gray'))
  )
```

`replaceData` *Replace data in an existing table*

Description

Replace the data object of a table output and avoid regenerating the full table, in which case the state of the current table will be preserved (sorting, filtering, and pagination) and applied to the table with new data.

Usage

```
replaceData(proxy, data, ..., resetPaging = TRUE, clearSelection = "all")

updateFilters(proxy, data)
```

Arguments

`proxy`	a proxy object created by `dataTableProxy()`
`data`	the new data object to be loaded in the table
`...`	other arguments to be passed to `dataTableAjax()`
`resetPaging, clearSelection`	passed to `reloadData()`

Note

When you replace the data in an existing table, please make sure the new data has the same number of columns as the current data. When you have enabled column filters, you should also make sure the attributes of every column remain the same, e.g. factor columns should have the same or fewer levels, and numeric columns should have the same or smaller range, otherwise the filters may never be able to reach certain rows in the data, unless you explicitly update the filters with `updateFilters()`.

If the `ColReorder` extension is used, the new `data` must have column names that match the original data column names exactly.

styleInterval *Conditional CSS styles*

Description

A few helper functions for the formatStyle() function to calculate CSS styles for table cells based on the cell values. Under the hood, they just generate JavaScript and CSS code from the values specified in R.

Usage

```
styleInterval(cuts, values)

styleEqual(levels, values, default = NULL)

styleValue()

styleColorBar(data, color, angle = 90)

styleRow(rows, values, default = NULL)
```

Arguments

cuts	a vector of cut points (sorted increasingly)
values	a vector of CSS values
levels	a character vector of data values to be mapped (one-to-one) to CSS values
default	a string or NULL used as the the default CSS value for values other than levels. If NULL, the CSS value of non-matched cells will be left unchanged.
data	a numeric vector whose range will be used for scaling the table data from 0-100 before being represented as color bars. A vector of length 2 is acceptable here for specifying a range possibly wider or narrower than the range of the table data itself.
color	the color of the bars
angle	a number of degrees representing the direction to fill the gradient relative to a horizontal line and the gradient line, going counter-clockwise. For example, 90 fills right to left and -90 fills left to right.
rows	the Row Indexes (starting from 1) that applies the CSS style. It could be an integer vector or a list of integer vectors, whose length must be equal to the length of values, when values is not a scalar.

Details

The function styleInterval() maps intervals to CSS values. Its argument values must be of length n + 1 where n = length(cuts). The right-closed interval '(cuts[i - 1], cuts[i]]' is mapped to 'values[i]' for 'i = 2, 3, ..., n'; 'values[1]' is for the interval '(-Inf, cuts[1]]', and ‘values[n + 1]’ is for ‘(cuts[n], +Inf)’. You can think of the order of cuts and values using this diagram: ‘-Inf -> values[1] -> cuts[1] -> values[2] -> cuts[2] -> ... -> values[n] -> cuts[n] -> values[n + 1] -> +Inf’.

The function styleEqual() maps data values to CSS values in the one-to-one manner, i.e. values[i] is used when the table cell value is levels[i].

The function styleColorBar() can be used to draw background color bars behind table cells in a column, and the width of bars is proportional to the column values.

The function styleValue() uses the column value as the CSS values.

The function styleRow() applies the CSS values based on Row Indexes. This only works expected in the client-side processing mode, i.e., server = FALSE.

tableHeader *Generate a table header or footer from column names*

Description

Convenience functions to generate a table header (‘<thead></thead>’) or footer (‘<tfoot></tfoot>’) given the column names. They are basically wrappers of htmltools::tags$th applied to the column names.

Usage

```
tableHeader(names, escape = TRUE)

tableFooter(names, escape = TRUE)
```

Arguments

names	a character vector of the column names of the table (if it is an object with column names, its column names will be used instead)
escape	whether to escape the names (see datatable)

Value

A tag object generated by htmltools::tags.

Examples

```
library(DT)
tableHeader(iris)  # or equivalently,
tableHeader(colnames(iris))
tableFooter(iris)  # footer

library(htmltools)
tags$table(tableHeader(iris), tableFooter(iris))
```

Index