Package 'echarty'

January 15, 2025

Title Minimal R/Shiny Interface to JavaScript Library 'ECharts'

Date 2025-01-14

Version 1.6.6

Description Deliver the full functionality of 'ECharts' with minimal overhead. 'echarty' users build R lists for 'ECharts' API. Lean set of powerful commands.

Depends R (>= 4.1.0)

Imports htmlwidgets, dplyr (>= 0.7.0), data.tree (>= 1.0.0),

Suggests htmltools (>= 0.5.0), shiny (>= 1.7.0), jsonlite, crosstalk, testthat (>= 3.0.0), sf, leaflet (>= 2.2.0), knitr, rmarkdown

RoxygenNote 7.3.2

License Apache License (>= 2.0)

URL https://helgasoft.github.io/echarty/

BugReports https://github.com/helgasoft/echarty/issues/

Encoding UTF-8

Language en-US

VignetteBuilder rmarkdown, knitr

NeedsCompilation no

Author Larry Helgason [aut, cre] (initial code from John Coene's library echarts4r)

Maintainer Larry Helgason <larry@helgasoft.com>

Repository CRAN

Date/Publication 2025-01-15 22:50:02 UTC

Contents

`- Introduction -` *echarty*

Description

echarty

Details

Description:

echarty provides a lean interface between R and Javascript library ECharts. We encourage users to follow the original ECharts API documentation to construct charts with echarty. Main command **ec.init** can set multiple native ECharts options to build a chart. The benefits - learn a very limited set of commands, and enjoy the **full functionality** of ECharts.

Package Conventions:

pipe-friendly - supports both %>% and |> commands have three prefixes to help with auto-completion:

- **ec.** for general functions, like *ec.init*
- **ecs.** for Shiny functions, like *ecs.output*
- **ecr.** for rendering functions, like *ecr.band*

Events:

For event handling in Shiny see sample code in eshiny.R, run as `demo(eshiny)`. Echarty has three built-in event callbacks - *click*, *mouseover*, *mouseout*. All other ECharts events could be initialized through `p$x$capture`. Another option is to use `p$x$on` with JavaScript handlers, see code in examples.

Widget x parameters:

These are *htmlwidget* and *ECharts* initialization parameters supported by echarty. There are code samples for most of them in examples:

- capture = event name(s), to monitor events, usually in Shiny
- on = define JavaScript code for event handling, see ECharts
- registerMap = define a map from a geoJSON file, see ECharts
- group = group-name of a chart, see ECharts
- connect = command to connect charts with same group-name, see ECharts
- locale = EN(default) or ZH, set from *locale* parameter of *ec.init*, see ECharts.
- renderer = *canvas*(default) or *svg*, set from *renderer* in *ec.init*, see ECharts.
- jcode = custom JavaScript code to execute, set from *js* parameter of *ec.init*

R vs Javascript numbering:

R language counting starts from 1. Javascript (JS) counting starts from 0. *ec.init* supports R-counting of indexes (ex. encode) and dimension (ex. visualMap). *ec.upd* requires indexes and dimensions to be set with JS-counting.

Javascript built-in functions:

To allow access to charts from JS. *ec_chart(id)* - get the chart object by id (former get_e_charts) *ec_option(id)* - get the chart's option object by id (former get_e_charts_opt) Parameter *id* could be the internal variable *echwid*, or the value set through *ec.init* parameter *elementId*. See demo code in examples

Code examples:

Here is the complete list of sample code **locations**:

- website gallery
- demo examples
- Shiny code is in eshiny.R, run with `demo(eshiny)`
- demos on RPubs
- searchable gists
- answers to Github issues
- code in Github tests
- command examples, like in *?ec.init*

Global Options:

Options are set with R command options. Echarty uses the following options:

- echarty.theme = name of theme file, without extension, from folder `/inst/themes`
- echarty.font = font family name
- echarty.urlTiles = tiles URL template for leaflet maps

```
# set/get global options
options('echarty.theme'='jazz') # set
getOption('echarty.theme')      # get
#> [1] "jazz"
options('echarty.theme'=NULL)   # remove
```

`ec.clmn` *Data column format*

Description

Helper function to display/format data column(s) by index or name

Usage

```
ec.clmn(col = NULL, ..., scale = 1)
```

Arguments

`col`	A single column index(number) or column name(quoted string), or a sprintf string template for multiple columns. NULL(default) for charts with single values like tree, pie. 'json' to display tooltip with all available values to choose from. 'log' to write all values in the JS console (F12) for debugging. Can contain JS function starting with *'function('* or *'(x) => {'*.
`...`	Comma separated column indexes or names, only when *col* is *sprintf*. This allows formatting of multiple columns, as for a tooltip.
`scale`	A positive number, multiplier for numeric columns. When scale is 0, all numeric values are rounded.

Details

This function is useful for attributes like formatter, color, symbolSize, label.
Column indexes are counted in R and start with 1.
Omit *col* or use index -1 for single values in tree/pie charts, *axisLabel.formatter* or *valueFormatter*.
See ec.data dendrogram example.
Column indexes are decimals for combo charts with multiple series, see ecr.band example. The whole number part is the serie index, the decimal part is the column index inside.
col as sprintf has the same placeholder %@ for both column indexes or column names.
col as sprintf can contain double quotes, but not single or backquotes.
Template placeholders with formatting:

- %@ will display column value as-is.
- *%L*@ will display a number in locale format, like '12,345.09'.
- *%LR*@ rounded number in locale format, like '12,345'.

- *%R@* rounded number, like '12345'.
- *%R2@* rounded number, two digits after decimal point.
- *%M@* marker in series' color.
  For *trigger='axis'* (multiple series) one can use decimal column indexes.
  See definition above and example below.

Value

A JavaScript code string (usually a function) marked as executable, see JS.

Examples

```
library(dplyr)
tmp <- data.frame(Species = as.vector(unique(iris$Species)),
                  emoji = c('A','B','C'))
df <- iris |> inner_join(tmp)      # add 6th column emoji
df |> group_by(Species) |> ec.init(
  series.param= list(label= list(show= TRUE, formatter= ec.clmn('emoji'))),
  tooltip= list(formatter=
    # with sprintf template + multiple column indexes
    ec.clmn('%M@ species <b>%@</b><br>s.len <b>%@</b><br>s.wid <b>%@</b>', 5,1,2))
)

# tooltip decimal indexes work with full data sets (no missing/partial data)
ChickWeight |> mutate(Chick=as.numeric(Chick)) |> filter(Chick>47) |> group_by(Chick) |>
ec.init(
  tooltip= list(trigger='axis',
                formatter= ec.clmn("48: %@<br>49: %@<br>50: %@", 1.1, 2.1, 3.1)),
  xAxis= list(type='category'), legend= list(formatter= 'Ch.{name}'),
  series.param= list(type='line', encode= list(x='Time', y='weight')),
)
```

`ec.data` *Data helper*

Description

Make data lists from a data.frame

Usage

```
ec.data(df, format = "dataset", header = FALSE, ...)
```

Arguments

`df` Required chart data as **data.frame**.
Except when format is *dendrogram*, then df is a **list**, result of hclust function.

`format` Output list format

- **dataset** = list to be used in dataset (default), or in series.data (without header).
- **values** = list for customized series.data
- **names** = named lists useful for named data like sankey links.
- **dendrogram** = build series data for Hierarchical Clustering dendrogram
- **treePC** = build series data for tree charts from parent/children data.frame
- **treeTT** = build series data for tree charts from data.frame like Titanic.
- **boxplot** = build dataset and source lists, see Details

`header` for dataset, to include the column names or not, default TRUE. Set it to FALSE for series.data.

`...` optional parameters
Optional parameters for **boxplot** are:

- *layout* = 'h' for horizontal(default) or 'v' for vertical layout
- *outliers* boolean to add outlier points (default FALSE)
- *jitter* value for jitter of numerical values in second column, default 0 (no scatter). Adds scatter series on top of boxplot.

Optional parameter for **names**:

- *nasep* = single character name separator for nested lists, see Examples. Purpose is to facilitate conversion from *data.frame* to nested named lists.

Details

`format='boxplot'` requires the first two *df* columns as:
column for the non-computational categorical axis
column with (numeric) data to compute the five boxplot values
Additional grouping is supported on a column after the second. Groups will show in the legend, if enabled.
Returns a `list(dataset, series, xAxis, yAxis)` to set params in ec.init. Make sure there is enough data for computation, 4+ values per boxplot.

`format='treeTT'` expects data.frame *df* columns *pathString,value,(optional itemStyle)* for FromDataFrameTable.
It will add column 'pct' with value percentage for each node. See example below.

Value

A list for *dataset.source*, *series.data* or other lists:
For boxplot - a named list, see Details and Examples
For dendrogram & treePC - a tree structure, see format in tree data

See Also

some live code samples

Examples

```
library(dplyr)
ds <- iris |> relocate(Species) |>
 ec.data(format= 'boxplot', jitter= 0.1, layout= 'v')
ec.init(
  dataset= ds$dataset, series= ds$series, xAxis= ds$xAxis, yAxis= ds$yAxis,
  legend= list(show= TRUE), tooltip= list(show= TRUE)
)

hc <- hclust(dist(USArrests), "complete")
ec.init(preset= FALSE,
  series= list(list(
    type= 'tree', orient= 'TB', roam= TRUE, initialTreeDepth= -1,
    data= ec.data(hc, format='dendrogram'),
    layout= 'radial', # symbolSize= ec.clmn(scale= 0.33),
    ## exclude added labels like 'pXX', leaving only the originals
    label= list(formatter= htmlwidgets::JS(
      "function(n) { out= /p\\d+/.test(n.name) ? '' : n.name; return out;}"))
  ))
)

# build required pathString,value and optional itemStyle columns
df <- as.data.frame(Titanic) |> rename(value= Freq) |> mutate(
  pathString= paste('Titanic\nSurvival', Survived, Age, Sex, Class, sep='/'),
 itemStyle= case_when(Survived=='Yes' ~"color='green'", TRUE ~"color='LightSalmon'")) |>
 select(pathString, value, itemStyle)
ec.init(
  series= list(list(
  data= ec.data(df, format='treeTT'),
  type= 'tree', symbolSize= ec.clmn("(x) => {return Math.log(x)*10}")
  )),
  tooltip= list(formatter= ec.clmn('%@<br>%@%','value','pct'))
)

# column itemStyle_color will become itemStyle= list(color=...) in data list
# attribute names separator (nasep) is "_"
df <- data.frame(name= c('A','B','C'), value= c(1,2,3),
```

```
    itemStyle_color= c('chartreuse','lightblue','pink'),
    itemStyle_decal_symbol= c('rect','diamond','none'),
    emphasis_itemStyle_color= c('darkgreen','blue','red')
)
ec.init(series.param= list(type='pie', data= ec.data(df, 'names', nasep='_')))
```

`ec.fromJson` *JSON to chart*

Description

Convert JSON string or file to chart

Usage

```
ec.fromJson(txt, ...)
```

Arguments

`txt`	Could be one of the following: class *url*, like `url('https://serv.us/cars.txt')` class *file*, like `file('c:/temp/cars.txt','rb')` class *json*, like `ec.inspect(p)`, for options or full class *character*, JSON string with options only, see example below
`...`	Any attributes to pass to internal ec.init when *txt* is options only

Details

txt could be either a list of options (x$opts) to be set by setOption,
OR an entire *htmlwidget* generated thru ec.inspect with *target='full'*.
The latter imports all JavaScript functions defined by the user.

Value

An *echarty* widget.

Examples

```
txt <- '{
   "xAxis": { "data": ["Mon", "Tue", "Wed"]}, "yAxis": { },
   "series": { "type": "line", "data": [150, 230, 224] } }'
ec.fromJson(txt)  # text json
# outFile <- 'c:/temp/cars.json'
# cars |> ec.init() |> ec.inspect(target='full', file=outFile)
# ec.fromJson(file(outFile, 'rb'))
# ec.fromJson(url('http://localhost/echarty/cars.json'))

ec.fromJson('https://helgasoft.github.io/echarty/test/pfull.json')
```

`ec.init` *Initialize command*

Description

Required to build a chart. In most cases this will be the only command necessary.

Usage

```
ec.init(
  df = NULL,
  preset = TRUE,
  ctype = "scatter",
  ...,
  series.param = NULL,
  tl.series = NULL,
  width = NULL,
  height = NULL
)
```

Arguments

`df`	Optional data.frame to be preset as dataset, default NULL By default the first column is for X values, second column is for Y, and third is for Z when in 3D. Best practice is to have the grouping column placed last. Grouping column cannot be used as axis. Timeline requires a *grouped data.frame* to build its options. If grouping is on multiple columns, only the first one is used to determine settings.
`preset`	Boolean (default TRUE). Build preset attributes like dataset, series, xAxis, yAxis, etc. When preset is FALSE, these attributes need to be set explicitly.
`ctype`	Chart type, default is 'scatter'. Could be set in *series.param* instead.
`...`	Optional widget attributes. See Details.
`series.param`	Additional attributes for single preset series, default is NULL. Defines a **single** series for both non-timeline and timeline charts. **Multiple** series should be defined directly with *series=list(list(type=...),list(type=...))* or added with ec.upd.
`tl.series`	Deprecated, use *timeline* and *series.param* instead.
`width, height`	Optional valid CSS unit (like `'100%'`, `'500px'`, `'auto'`) or a number, which will be coerced to a string and have `'px'` appended.

Details

Command *ec.init* creates a widget with createWidget, then adds some ECharts features to it.
Numerical indexes for series,visualMap,etc. are R-counted (1,2...)

Presets

When data.frame **df** is present, a dataset is preset.
When **df** is grouped and *ctype* is not NULL, more datasets with legend and series are also preset.
Plugin '3D' (load='3D') is required for GL series like *scatterGL, linesGL*, etc.
Plugins 'leaflet' and 'world' preset *center* to the mean of all coordinates from **df**.
Users can delete or overwrite any presets as needed.

Widget attributes

Optional echarty widget attributes include:

- elementId - Id of the widget, default is NULL(auto-generated)
- load - name(s) of plugin(s) to load. A character vector or comma-delimited string. default NULL.
- ask - prompt user before downloading plugins when *load* is present, FALSE by default
- js - single string or a vector with JavaScript expressions to evaluate.
  single: exposed *chart* object (most common)
  vector: see code in examples
  First expression evaluated before initialization, exposed object *window*
  Second is evaluated with exposed object *opts*.
  Third is evaluated with exposed *chart* object after *opts* set.
- renderer - 'canvas'(default) or 'svg'
- locale - 'EN'(default) or 'ZH'. Use predefined or custom like so.
- useDirtyRect - enable dirty rectangle rendering or not, FALSE by default, see here

Built-in plugins

- leaflet - Leaflet maps with customizable tiles, see source
- world - world map with country boundaries, see source
- lottie - support for lotties
- ecStat - statistical tools, seeecharts-stat
- custom - renderers for ecr.band and ecr.ebars
  Plugins with one-time installation:

- 3D - support for 3D charts and WebGL acceleration, see source and docs
  This plugin is auto-loaded when 3D/GL axes/series are detected.
- liquid - liquid fill, see source
- gmodular - graph modularity, see source
- wordcloud - cloud of words, see source
  or install your own third-party plugins.

Crosstalk
Parameter *df* should be of type SharedData, see more info.
Optional parameter *xtKey*: unique ID column name of data frame *df*. Must be same as *key* parameter used in *SharedData$new()*. If missing, a new column *XkeyX* will be appended to df.
Enabling *crosstalk* will also generate an additional dataset called *Xtalk* and bind the **first series** to it.

Timeline
Defined by *series.param* for the options series and a *timeline* list for the actual control. A grouped *df* is required, each group providing data for one option serie. Timeline data and options will be preset for the chart.
Each option title can include the current timeline item by adding a placeholder '%@' in title$text. See example below.
Another preset is *encode(x=1,y=2,z=3)*, which are the first 3 columns of *df*. Parameter *z* is ignored in 2D. See Details below.
Optional attribute *groupBy*, a *df* column name, can create series groups inside each timeline option. Options/timeline for hierarchical charts like graph,tree,treemap,sankey have to be built directly, see example.

Optional series attribute encode defines which columns to use for the axes, depending on chart type and coordinate system:

- set *x* and *y* for coordinateSystem *cartesian2d*
- set *lng* and *lat* for coordinateSystem *geo* and *scatter* series
- set *value* and *name* for coordinateSystem *geo* and *map* series
- set *radius* and *angle* for coordinateSystem *polar*
- set *value* and *itemName* for *pie* chart.

Example: `encode(x='col3', y='col1')` binds xAxis to *df* column 'col3'.

Value

A widget to plot, or to save and expand with more features.

Examples

```
 # basic scatter chart from a data.frame, using presets
cars |> ec.init()

 # grouping, tooltips, formatting
iris |> dplyr::group_by(Species) |>
ec.init(        # init with presets
  tooltip= list(show= TRUE),
  series.param= list(
    symbolSize= ec.clmn('Petal.Width', scale=7),
    tooltip= list(formatter= ec.clmn('Petal.Width: %@', 'Petal.Width'))
  )
)

data.frame(n=1:5) |> dplyr::group_by(n) |> ec.init(
  title= list(text= "gauge #%@"),
  timeline= list(show=TRUE, autoPlay=TRUE),
  series.param= list(type='gauge', max=5)
)
```

`ec.inspect` *Chart to JSON*

Description

Convert chart to JSON string

Usage

```
ec.inspect(wt, target = "opts", ...)
```

Arguments

`wt`	An `echarty` widget as returned by ec.init
`target`	type of resulting value: 'opts' - the htmlwidget *options* as JSON (default) 'full' - the *entire* htmlwidget as JSON 'data' - info about chart's embedded data (char vector)
`...`	Additional attributes to pass to toJSON 'file' - optional file name to save to when target='full'

Details

Must be invoked or chained as last command.
target='full' will export all JavaScript custom code, ready to be used on import.
See also ec.fromJson.

Value

A JSON string, except when `target` is 'data' - then a character vector.

Examples

```
# extract JSON
json <- cars |> ec.init() |> ec.inspect()
json

# get from JSON and modify plot
ec.fromJson(json) |> ec.theme('macarons')
```

`ec.paxis` *Parallel Axis*

Description

Build 'parallelAxis' for a parallel chart

Usage

```
ec.paxis(dfwt = NULL, cols = NULL, minmax = TRUE, ...)
```

Arguments

`dfwt`	An echarty widget OR a data.frame(regular or grouped)
`cols`	A string vector with columns names in desired order
`minmax`	Boolean to add max/min limits or not, default TRUE
`...`	Additional attributes for parallelAxis.

Details

This function could be chained to *ec.init* or used with a *data.frame*

Value

A list, see format in parallelAxis.

Examples

```
iris |> dplyr::group_by(Species) |>    # chained
ec.init(ctype= 'parallel', series.param= list(lineStyle= list(width=3))) |>
ec.paxis(cols= c('Petal.Length','Petal.Width','Sepal.Width'))

mtcars |> ec.init(ctype= 'parallel',
   parallelAxis= ec.paxis(mtcars, cols= c('gear','cyl','hp','carb'), nameRotate= 45),
   series.param= list(smooth= TRUE)
)
```

`ec.plugjs` *Install Javascript plugin from URL source*

Description

Install Javascript plugin from URL source

Usage

```
ec.plugjs(wt = NULL, source = NULL, ask = FALSE)
```

Arguments

`wt`	A widget to add dependency to, see createWidget
`source`	URL or file:// of a Javascript plugin, file name suffix is '.js'. Default is NULL.
`ask`	Boolean, to ask the user to download source if missing. Default is FALSE.

Details

When *source* is URL, the plugin file is installed with an optional popup prompt.
When *source* is a file name (file://xxx.js), it is assumed installed and only a dependency is added.
When *source* is invalid, an error message will be written in the chart's title.
Called internally by ec.init. It is recommended to use *ec.init(load=...)* instead of *ec.plugjs*.

Value

A widget with JS dependency added if successful, otherwise input wt

Examples

```
# import map plugin and display two (lon,lat) locations
if (interactive()) {
  durl <- paste0('https://raw.githubusercontent.com/apache/echarts/',
           'master/test/data/map/js/china-contour.js')
  ec.init(  # load= durl,
    geo = list(map= 'china-contour', roam= TRUE),
```

```
    series.param = list(
      type= 'scatter', coordinateSystem= 'geo',
      symbolSize= 9, itemStyle= list(color= 'red'),
      data= list(list(value= c(113, 40)), list(value= c(118, 39))) )
  ) |>
  ec.plugjs(durl)
}
```

---

ec.theme *Themes*

Description

Apply a pre-built or custom coded theme to a chart

Usage

```
ec.theme(wt, name = "custom", code = NULL)
```

Arguments

wt	Required echarty widget as returned by ec.init
name	Name of existing theme file (without extension), or name of custom theme defined in code.
code	Custom theme as JSON formatted string, default NULL.

Details

Just a few built-in themes are included in folder `inst/themes`.
Their names are dark, gray, jazz, dark-mushroom and macarons.
The entire ECharts theme collection could be found here and files copied if needed.
To create custom themes or view predefined ones, visit this site.

Value

An echarty widget.

Examples

```
mtcars |> ec.init() |> ec.theme('dark-mushroom')
cars |> ec.init() |> ec.theme('mine', code=
  '{"color": ["green","#eeaa33"],
    "backgroundColor": "lemonchiffon"}')
```

ec.upd *Update option lists*

Description

Chain commands after ec.init to add or update chart items

Usage

```
ec.upd(wt, ...)
```

Arguments

wt	An echarty widget
...	R commands to add/update chart option lists

Details

ec.upd makes changes to a chart already set by ec.init.
It should be always piped(chained) after ec.init.
All numerical indexes for series,visualMap,etc. are JS-counted starting at 0.

Examples

```
library(dplyr)
df <- data.frame(x= 1:30, y= runif(30, 5, 10), cat= sample(LETTERS[1:3],size=30,replace=TRUE)) |>
   mutate(lwr= y-runif(30, 1, 3), upr= y+runif(30, 2, 4))
band.df <- df  |> group_by(cat) |> group_split()

df |> group_by(cat) |>
ec.init(load='custom', ctype='line',
        xAxis=list(data=c(0,unique(df$x)), boundaryGap=FALSE) ) |>
ec.upd({
  for(ii in 1:length(band.df))   # add bands to their respective groups
    series <- append(series,
      ecr.band(band.df[[ii]], 'lwr', 'upr', type='stack', smooth=FALSE,
      name= unique(band.df[[ii]]$cat), areaStyle= list(color=c('blue','green','yellow')[ii])) )
})
```

`ec.util` *Utility functions*

Description

tabset, table layout, support for GIS shapefiles through library 'sf'

Usage

```
ec.util(..., cmd = "sf.series", js = NULL, event = "click")
```

Arguments

`...`	Optional parameters for the command for *sf.series* - see points, polylines, polygons(itemStyle). for *tabset* parameters should be in format *name1=chart1, name2=chart2*, see example
`cmd`	utility command, see Details
`js`	optional JavaScript function, default is NULL.
`event`	optional event name for cmd='morph'.

Details

cmd = 'sf.series'
Build *leaflet* or geo map series from shapefiles.
Supported types: POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, MULTIPOLYGON
Coordinate system is *leaflet*(default), *geo* or *cartesian3D* (for POINT(xyz))
Limitations:
polygons can have only their name in tooltip,
assumes Geodetic CRS is WGS 84, for conversion use st_transform with *crs=4326*.
Parameters:
df - value from st_read
nid - optional column name for name-id used in tooltips
cs - optional *coordinateSystem* value, default 'leaflet'
verbose - optional, print shapefile item names in console
Returns a list of chart series
cmd = 'sf.bbox'
Returns JavaScript code to position a map inside a bounding box from st_bbox, for leaflet only.
cmd = 'sf.unzip'
Unzips a remote file and returns local file name of the unzipped .shp file
url - URL of remote zipped shapefile
shp - optional name of .shp file inside ZIP file if multiple exist. Do not add file extension.
Returns full name of unzipped .shp file, or error string starting with 'ERROR'
cmd = 'geojson'

Custom series list from geoJson objects
geojson - object from fromJSON
cs - optional *coordinateSystem* value, default 'leaflet'
ppfill - optional fill color like '#F00', OR NULL for no-fill, for all Points and Polygons
nid - optional feature property for item name used in tooltips
... - optional custom series attributes like *itemStyle*
Can display also geoJson *feature properties*: color; lwidth, ldash (lines); ppfill, radius (points)
cmd = 'layout'
Multiple charts in table-like rows/columns format
... - List of charts
title - optional title for the entire set
rows - optional number of rows
cols - optional number of columns
Returns a container div in rmarkdown, otherwise browsable.
For 3-4 charts one would use multiple series within a grid.
For greater number of charts *ec.util(cmd='layout')* comes in handy
cmd = 'tabset'
... - a list name/chart pairs like *n1=chart1, n2=chart2*, each tab may contain a chart.
tabStyle - tab style string, see default *tabStyle* variable in the code
Returns A) tagList of tabs when in a pipe without '...' params, see example
Returns B) browsable when '...' params are provided by user
Please note that sometimes those tabsets do not merge well inside advanced web pages.
cmd = 'button'
UI button to execute a JS function,
text - the button label
js - the JS function string
... - optional parameters for the rect element
Returns a graphic.elements-rect element.
cmd = 'morph'
... - a list of charts or chart option lists
event - name of event for switching charts. Default is *click*.
Returns a chart with ability to morph into other charts
cmd = 'fullscreen'
A toolbox feature to toggle fullscreen on/off. Works in a browser, not in RStudio.
cmd = 'rescale'
v - input vector of numeric values to rescale
t - target range c(min,max), numeric vector of two
cmd = 'level'
Calculate vertical levels for timeline *line* charts, returns a numeric vector
df - data.frame with *from* and *to* columns
from - name of 'from' column
to - name of 'to' column

Examples

```
library(dplyr)
if (interactive()) {  # comm.out: Cran Fedora errors about some 'browser'
  library(sf)
```

```
  fname <- system.file("shape/nc.shp", package="sf")
  nc <- as.data.frame(st_read(fname))
  ec.init(load= c('leaflet', 'custom'),  # load custom for polygons
     js= ec.util(cmd= 'sf.bbox', bbox= st_bbox(nc$geometry)),
    series= ec.util(cmd= 'sf.series', df= nc, nid= 'NAME', itemStyle= list(opacity=0.3)),
     tooltip= list(formatter= '{a}')
  )

  htmltools::browsable(
    lapply(iris |> group_by(Species) |> group_split(),
           function(x) {
     x |> ec.init(ctype= 'scatter', title= list(text= unique(x$Species)))
           }) |>
    ec.util(cmd= 'tabset')
  )

  p1 <- cars |> ec.init(grid= list(top=26), height=333)  # move chart up
  p2 <- mtcars |> arrange(mpg) |> ec.init(height=333, ctype='line')
  ec.util(cmd= 'tabset', cars= p1, mtcars= p2)


  cars |> ec.init(
    graphic = list(
      ec.util(cmd='button', text='see type', right='center', top=20,
        js="function(a) {op=ec_option(echwid); alert(op.series[0].type);}")
    )
  )

  lapply(list('dark','macarons','gray','jazz','dark-mushroom'),
    function(x) cars |> ec.init(grid= list(bottom=0)) |> ec.theme(x) ) |>
  ec.util(cmd='layout', cols= 2, title= 'my layout')
}

colors <- c("blue","red","green")
cyls <- as.character(sort(unique(mtcars$cyl)))
sers <- lapply(mtcars |> group_by(cyl) |> group_split(), \(x) {
  cyl <- as.character(unique(x$cyl))
  list(type='scatter', id=cyl, dataGroupId=cyl,
       data= ec.data(x |> select(mpg,hp)),
       universalTransition= TRUE)
})
oscatter <- list(
  title= list(subtext='click points to morph'),
  color= colors, tooltip= list(show=TRUE),
  xAxis= list(scale=TRUE, name='mpg'), yAxis= list(scale=TRUE, name='hp'),
  series= sers
)
opie <- list(
  title= list(text= 'Average hp'),
  color= colors, tooltip= list(show=TRUE),
  series= list(list(
    type= 'pie', label= list(show=TRUE), colorBy= 'data',
    data= ec.data(mtcars |> group_by(cyl) |> summarize(value= mean(hp)) |>
```

```
      mutate(groupId= as.character(cyl), name= as.character(cyl)),'names'),
    universalTransition= list(enabled=TRUE, seriesKey= cyls)
  ))
)
ec.util(cmd='morph', oscatter, opie)
```

`ecr.band` *Area band*

Description

A 'custom' serie with lower and upper boundaries

Usage

```
ecr.band(df = NULL, lower = NULL, upper = NULL, type = "polygon", ...)
```

Arguments

`df`	A data.frame with lower and upper numerical columns and first column with X coordinates.
`lower`	The column name of band's lower boundary (string).
`upper`	The column name of band's upper boundary (string).
`type`	Type of rendering • 'polygon' - by drawing a polygon as polyline from upper/lower points (default) • 'stack' - by two stacked lines
`...`	More attributes for serie

Details

- type='polygon': coordinates of the two boundaries are chained into one polygon.
  xAxis type could be 'category' or 'value'.
  Set fill color with attribute *color*.
- type='stack': two *stacked* lines are drawn, the lower with customizable areaStyle.
  xAxis type should be 'category' !
  Set fill color with attribute *areaStyle$color*.
  Optional tooltip formatter available in *band[[1]]$tipFmt*.

Optional parameter *name*, if given, will show up in legend. Legend merges all series with same name into one item.

Value

A list of **one serie** when type='polygon', or list of **two series** when type='stack'

Examples

```
set.seed(222)
df <- data.frame( x = 1:10, y = round(runif(10, 5, 10),2)) |>
  dplyr::mutate(lwr= round(y-runif(10, 1, 3),2), upr= round(y+runif(10, 2, 4),2) )
banda <- ecr.band(df, 'lwr', 'upr', type='stack', name='stak', areaStyle= list(color='green'))
#banda <- ecr.band(df, 'lwr', 'upr', type='polygon', name='poly1')

df |> ec.init( load='custom', # polygon only
  legend= list(show= TRUE),
  xAxis= list(type='category', boundaryGap=FALSE), # stack
  #xAxis= list(scale=T, min='dataMin'),            # polygon
  series= append(
    list(list(type='line', color='blue', name='line1')),
    banda
  ),
  tooltip= list(trigger='axis', formatter= banda[[1]]$tipFmt)
)
```

`ecr.ebars` *Error bars*

Description

Custom series to display error-bars for scatter, bar or line series

Usage

```
ecr.ebars(wt, encode = list(x = 1, y = c(2, 3, 4)), hwidth = 6, ...)
```

Arguments

`wt`	An echarty widget to add error bars to, see ec.init.
`encode`	Column selection for both axes (x & y) as vectors, see encode
`hwidth`	Half-width of error bar in pixels, default is 6.
`...`	More parameters for custom serie

Details

Command should be called after *ec.init* where main series are set.
ecr.ebars are custom series, so *ec.init(load='custom')* is required.
Horizontal and vertical layouts supported, just switch *encode* values *x* and *y* for both for series and ecr.ebars.
Have own default tooltip format showing *value, high & low*.
Grouped bar series are supported.
Non-grouped series could be shown with formatter *riErrBarSimple* instead of *ecr.ebars*. This is limited to vertical only, see example below.

Other limitations:
manually add axis type='category' when needed
error bars cannot have own name when data is grouped
legend select/deselect will not re-position grouped error bars

Value

A widget with error bars added if successful, otherwise the input widget

Examples

```
library(dplyr)
df <- mtcars |> group_by(cyl,gear) |> summarise(avg.mpg= round(mean(mpg),2)) |>
  mutate(low = round(avg.mpg-cyl*runif(1),2),
         high= round(avg.mpg+cyl*runif(1),2))
ec.init(df, load= 'custom', ctype= 'bar',
      xAxis= list(type='category'), tooltip= list(show=TRUE)) |>
ecr.ebars(encode= list(y=c('avg.mpg','low','high'), x='gear'))
#ecr.ebars(encode= list(y=c(3,4,5), x=2))  # ok: data indexes

# same but horizontal
ec.init(df, load= 'custom',
  yAxis= list(type='category'), tooltip= list(show=TRUE),
  series.param= list(type='bar', encode= list(x='avg.mpg', y='gear') )) |>
ecr.ebars(encode= list(x=c('avg.mpg','low','high'), y='gear'))

# ----- riErrBarSimple ------
df <- mtcars |> mutate(name= row.names(mtcars), hi= hp-drat*3, lo= hp+wt*3) |>
  filter(cyl==4) |> select(name,hp,hi,lo)
ec.init(df, load= 'custom', legend= list(show=TRUE)) |>
ec.upd({ series <- append(series, list(
  list(type= 'custom', name= 'error',
    data= ec.data(df |> select(name,hi,lo)),
    renderItem= htmlwidgets::JS('riErrBarSimple')
  )))
})
```

`ecs.exec` *Shiny: Execute a proxy command*

Description

Once chart changes had been made, they need to be sent back to the widget for display

Usage

```
ecs.exec(proxy, cmd = "p_merge")
```

Arguments

proxy	A ecs.proxy object
cmd	Name of command, default is *p_merge* The proxy commands are: *p_update* - add new series and axes *p_merge* - modify or add series features like style,marks,etc. *p_replace* - replace entire chart *p_del_serie* - delete a serie by index or name *p_del_marks* - delete marks of a serie *p_append_data* - add data to existing series *p_dispatch* - send action commands, see documentation

Value

A proxy object to update the chart.

See Also

ecs.proxy, ecs.render, ecs.output
Read about event handling in – Introduction –, or from examples.

Examples

```
if (interactive()) {
 # run with  demo(eshiny, package='echarty')
}
```

---

ecs.output *Shiny: UI chart*

Description

Placeholder for a chart in Shiny UI

Usage

```
ecs.output(outputId, width = "100%", height = "400px")
```

Arguments

outputId	Name of output UI element.
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.

Value

An output or render function that enables the use of the widget within Shiny applications.

See Also

ecs.exec for example, shinyWidgetOutput for return value.

`ecs.proxy` *Shiny: Create a proxy*

Description

Create a proxy for an existing chart in Shiny UI. It allows to add, merge, delete elements to a chart without reloading it.

Usage

```
ecs.proxy(id)
```

Arguments

`id`	Target chart id from the Shiny UI.

Value

A proxy object to update the chart.

See Also

ecs.exec for example.

`ecs.render` *Shiny: Plot command to render chart*

Description

This is the initial rendering of a chart in the UI.

Usage

```
ecs.render(wt, env = parent.frame(), quoted = FALSE)
```

Arguments

`wt`	An `echarty` widget to generate the chart.
`env`	The environment in which to evaluate `expr`.
`quoted`	Is `expr` a quoted expression? default FALSE.

Value

An output or render function that enables the use of the widget within Shiny applications.

See Also

ecs.exec for example, shinyRenderWidget for return value.

Index