

Package ‘echarty’

May 8, 2026

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

Date 2026-04-03

Version 1.7.2

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 7.3.3

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 2026-04-04 07:40:02 UTC

Contents

– Introduction –	2
ec.clmn	4
ec.data	5
ec.fromJson	8

ec.init	9
ec.inspect	13
ec.paxis	14
ec.plugin	15
ec.registerMap	16
ec.theme	17
ec.upd	18
ec.util	19
ecr.band	22
ecr.ebars	23
ecs.exec	24
ecs.output	25
ecs.proxy	26
ecs.render	26
Index	28

- 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:

Event handling in **Shiny** is done through callbacks. See considerable sample code in [eshiny.R](#), run as `demo(eshiny)`. There are three built-in event callbacks - *click*, *mouseover*, *mouseout*. All other ECharts **events** could be initialized through `ec.init(capture=...)`. For event handling in R (without Shiny) use parameter `ec.init(on=...)` which expects JavaScript handlers. Search for 'event' in [code examples](#).

ECharts initialization parameters:

Chart initialization is performed by the `echarty::ec.init()` command. Here is full list of `ec.init` optional parameters:

- `ask`, `js`, `elementId`, `ctype`, `xtKey`, `dbg` are specific to `echarty`
- `theme`, `iniOpts`, `on`, `off`, `capture`, `group` belong to the ECharts **chart instance** object.
- `connect`, `disconnect`, `registerMap`, `registerTheme`, `registerLocale`, `registerCustomSeries` are commands of the global **ECharts object**.

There are R **code examples** for some of these parameters.

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`). All other contexts like `ec.upd` or `ecs.proxy` require JS-counting of indexes and dimensions.

Javascript built-in functions:

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

Column-to-style binding with encode:

ECharts **series encode** enables binding axes and tooltip to data columns. Echarty enhances this method for all **series data** parameters like `itemStyle`, `labels`, `emphasis`, etc. The bindings are set through `series$encode$data`. For instance `encode= list(data= list(value= c('xc', 'yc'), itemStyle= list(opacity= 'oc')))` would match columns `xc`, `yc`, `oc` to each item's value and opacity. The result is a new `series$data` added to the series. It permits to finely customize chart elements directly from data. Echarty has also an alternative tool, style-named columns with `ec.data(. .nasep)`, but `encode$data` offers more flexibility. It is not compatible with *timeline* however.

Code examples:

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

- website **gallery**
- collection of **code 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	Can contain one of several types of values: NULL(default) for charts with single values like tree, pie. a single column index(number) or column name(quoted string) a <code>sprintf</code> string template for multiple columns 'json' to display tooltip with all available values to choose from 'log' to write all values in the JS console (F12) for debugging. a string containing a JS function starting with <code>'function('</code> or <code>'(x) =>'</code> .
...	Comma separated column indexes or names, only when <i>col</i> is <i>sprintf</i> . 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	<p>Required chart data as data.frame. For format <i>dendrogram</i> df is a list, result of hclust function. For format <i>flame</i> df is an hierarchical list with name,value,children.</p>
format	<p>Output list format</p> <ul style="list-style-type: none"> • 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 • flame = build series data (lists of name,id,value) for hierarchy display by <i>renderItem</i> • 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 • borders = build geoJson string from map_data region borders, see Details
header	<p>for dataset, to include the column names or not, default TRUE. Set it to FALSE for series.data.</p>
...	<p>optional parameters Optional parameters for boxplot are:</p> <ul style="list-style-type: none"> • <i>layout</i> = 'h' for horizontal(default) or 'v' for vertical layout • <i>outliers</i> boolean to add outlier points (default FALSE) • <i>jitter</i> value for jitter of numerical values in second column, default 0 (no scatter). Adds scatter series on top of boxplot. <p>Optional parameter for names:</p> <ul style="list-style-type: none"> • <i>nasep</i> = single character name separator for nested lists, see Examples. Purpose is to facilitate conversion from <i>data.frame</i> to nested named lists. <p>Optional parameter for flame:</p> <ul style="list-style-type: none"> • <i>name</i> = name of subtree to search for.

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 [From-DataFrameTable](#).

It will add column 'pct' with value percentage for each node. See example below.

format='borders' expects *df* columns *long,lat,region,subregion* as in `ggplot2::map_data`.
 Result to be used as map in `ec.registerMap`. See borders code example in *examples.R*.
 This is a slow version for borders, another very fast one is offered as `echarty extra`, see website.

Value

A list for *dataset.source, series.data* or other lists:
 For boxplot - a named list, see Details and Examples
 For dendrogram, treePC, flame - 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 <i>url</i> , like <code>url('https://serv.us/cars.txt')</code> class <i>file</i> , like <code>file('c:/temp/cars.txt', 'rb')</code> class <i>json</i> , like <code>ec.inspect(p)</code> , for options or full class <i>character</i> , JSON string with options only, see example below
...	Any attributes to pass to internal <code>ec.init</code> when <i>txt</i> 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 a chart

Description

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

Usage

```
ec.init(
  df = NULL,
  preset = TRUE,
  ...,
  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.
...	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. Default type is 'scatter'. Multiple series need to be defined directly with <code>series=list(list(type=...),list(type=...))</code> or added with ec.upd .
tl.series	Deprecated, use <code>timeline</code> and <code>series.param</code> 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

A **dataset** is pre-set when data.frame **df** is present.

When **df** is grouped, more datasets with legend and series are also preset.

Axes for some charts are preset with name and type when suitable.

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, stored as `echwid` variable for JS)
- `load` - name(s) of plugin(s) to load. A character vector or comma-delimited string. default NULL.
- `ask` - boolean to prompt user before downloading plugins when `load` is present, default is FALSE.
Could also be string 'loadRemote' to load plugins remotely.
- `ctype` - alternative way of setting chart type name, default is 'scatter'.
- `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 with exposed objects *window* and *echarts*

Second is evaluated with exposed object *opts*.

Third is evaluated with exposed *chart* object after initialization with *opts* already set.

- *theme* - name of built-in theme to apply, or JSON object from *fromJSON*, see *opts* in [echarts.init](#)
- *iniOpts* - a list of initialization options, see *opts* in [echarts.init](#)
Defaults: *renderer*='canvas', *locale*='EN', *useDirtyRect*=FALSE
- *on,off,capture,group* - chart instance properties, namely:
on/off is a list of events to handle with JS, each in a list, see [chart.on](#) and example below
capture is a vector of event names to capture in Shiny, etc.
- *connect,disconnect,register,etc.* - see [echarts object](#) methods

Built-in plugins

- *leaflet* - Leaflet maps with customizable tiles, see [source](#)
- *world* - world map with country boundaries, see [source](#)
- *lottie* - support for <https://lottiefiles.com>
- *ecStat* - statistical tools, see [echarts-stat](#)
- *custom* - renderers for echarty plugins like [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.
- *gmodular* - graph modularity, see [source](#)
- *liquid* - liquid fill, see [source](#)
- *wordcloud* - cloud of words, see [source](#)
Note: the last three are being moved to the [official custom series](#).
OR install your own third-party plugins like *confetti*, see example below.

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 `crostalk` 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.

There is an advanced usage of `encode` when each series' item needs to be customized.

For example `encode= list(itemStyle= list(opacity='opac'))` will create series data where each series item's opacity comes from `df` column `'opac'`.

This binding feature is specific to `echarty` and does not exist in `ECharts`. See example below.

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()

# custom initialization options and theme
myth <- '{"color": ["green"], "backgroundColor": "lemonchiffon"}'
ec.init( cars,
  iniOpts= list(renderer= 'svg', width= '222px'),
  theme= jsonlite::fromJSON(myth),
  toolbox= list(feature= list(saveAsImage= list()))
)

# grouping, tooltips, formatting, events
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'))
  ),
  on= list( # events with Javascript handler
    list(event= 'legendselectchanged', handler= ec.clmn("(e) => alert('legend: '+e.name);"))
  )
)

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.init(
  series.param= list(
    renderItem= 'segmentedDoughnut', # v.6 from https://github.com/apache/echarts-custom-series
    itemPayload= list(segmentCount= 8, label= list(show=TRUE, formatter= '{c}/{b}', fontSize=35) ),
    data= list(5) )
)

ec.init(cars, js= 'confetti();', # js code executes on init
  load= 'https://cdn.jsdelivr.net/npm/canvas-confetti@1.9.4/dist/confetti.browser.min.js',
  ask= 'loadRemote',
  on= list(list(event= 'click', handler= ec.clmn('() => confetti()')) )
)

```

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 <i>htmlwidget options</i> as JSON (default) 'full' - the <i>entire htmlwidget</i> 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(series.param= list(type= 'parallel', lineStyle= list(width=3))) |>
ec.paxis(cols= c('Petal.Length', 'Petal.Width', 'Sepal.Width'))

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

 ec.pluginjs

Install Javascript plugin from URL source

Description

Install Javascript plugin from URL source

Usage

```
ec.pluginjs(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, whether to ask the user to download source if missing, default is FALSE. Could also be string 'loadRemote' to load plugins remotely.

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.pluginjs*.

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.pluginjs(durl)
}
```

ec.registerMap	<i>Register a geoJson map</i>
----------------	-------------------------------

Description

Read geoJSON file to be used in a map chart

Deprecated since v.1.7.0, use `ec.init(registerMap=...)` instead.

Usage

```
ec.registerMap(wt = NULL, name = "loadedMapName", data = NULL)
```

Arguments

wt	An echarty widget as returned by <code>ec.init</code> .
name	Name of map.
data	A string starting with <i>http</i> or <i>file</i> . URL strings ending with <i>.svg</i> are assumed to be SVG map files. Could also be a valid geoJSON or SVG text string. SVG strings start with either <i><?xml</i> or <i><svg</i> .

Details

This command replaces the manual setting through `p$$registerMap`.

It should always be piped after `ec.init`.

There should be one map series with attribute 'map' matching the name parameter.

Value

An *echarty* widget.

Examples

```
data.frame(name= c('Texas', 'California'), value= c(111, 222)) |>
ec.init( color= c('lightgray'), visualMap= list(min=111),
  series.param= list(type= 'map', map= 'usa')
) |>
ec.registerMap('usa', 'https://echarts.apache.org/examples/data/asset/geo/USA.json')
```

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 [theme-builder](#).
 See also alternative *registerTheme* in [ec.init](#).

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

<code>wt</code>	An echarty widget
<code>...</code>	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()
sband <- list()
for(ii in seq_along(band.df)) # build all bands
  sband <- append(sband,
    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]))
  )

df |> group_by(cat) |>
ec.init(load='custom', series.param= list(type='line'),
  xAxis=list(data=c(0,unique(df$x)), boundaryGap=FALSE) ) |>
ec.upd({ series <- append(series, sband) })
```

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

cmd	Utility command name, see Details.
...	Optional parameters for each command.
js	Optional JavaScript function, default is NULL.
event	Optional event name for cmd='morph', default is 'click'.

Details**cmd = 'sf.series'**

Build *leaflet* or *geo* map series from shapefiles.

Supported types: POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, MULTIPOLYGON

Coordinate system could be *leaflet*(default), *geo*, *cartesian2D* or *cartesian3D*(for POINT(xyz))

Limitations:

polygons can have only their name in tooltip, need *load='custom'* for rendering

assumes Geodetic CRS is WGS 84, for conversion use [st_transform](#) with *crs=4326*.

Parameters:

df - value from [st_read](#)

nid - optional column name used in tooltip formatter

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'
 ppsfill - 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); ppsfill, 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 of name/chart pairs like `n1=chart1, n2=chart2`, each tab may contain a chart, see example
 tabStyle - tab style string, see default `strTabStyle` variable in the code
 width - optional width size for the tabset, in CSS format, default is 100%
 Returns A) `browsable` when '...' params are provided
 Returns B) `tagList` of tabs when in a pipe (no '...' params)
 Please note that a maximum of five(5) tabs are supported by current `tabStyle`.

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}')
  )
}

if (interactive()) {
  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)

  lapply(list('dark', 'macarons', 'gray', 'dark-mushroom'),
    function(x) cars |> ec.init(grid= list(bottom=5, top=10)) |> ec.theme(x) ) |>
  ec.util(cmd='layout', cols= 2, title= 'Layout')
}

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);}")
  )
)

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)
})
osscatter <- list(
  title= list(text='Morph', left='center', 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)
    ))
)
ecr.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 <ul style="list-style-type: none"> '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=TRUE, 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', series.param= list(type='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 with 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,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 <i>p_merge</i> The proxy commands are: <i>p_update</i> - add new series and axes <i>p_merge</i> - modify or add series features like style,marks,etc. <i>p_replace</i> - replace entire chart <i>p_del_serie</i> - delete a serie by index or name <i>p_del_marks</i> - delete marks of a serie <i>p_append_data</i> - add data to existing series <i>p_dispatch</i> - 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

- Introduction -, 2, 25

browsable, 20

createWidget, 10, 15

div, 20

ec.clmn, 4

ec.data, 4, 5

ec.fromJson, 8, 14

ec.init, 7, 8, 9, 13, 15–18, 23

ec.inspect, 8, 13

ec.paxis, 14

ec.pluginjs, 15

ec.registerMap, 7, 16

ec.theme, 17

ec.upd, 10, 18

ec.util, 19

ecr.band, 4, 11, 22

ecr.ebars, 11, 23

ecs.exec, 24, 26, 27

ecs.output, 25, 25

ecs.proxy, 25, 26

ecs.render, 25, 26

FromDataFrameTable, 7

fromJSON, 19

hclust, 6

jitter, 6

JS, 5

SharedData, 11

shinyRenderWidget, 27

shinyWidgetOutput, 26

sprintf, 4

st_bbox, 19

st_read, 19

st_transform, 19

tagList, 20

toJSON, 14