# Package 'leaflet'

June 24, 2015

**Type** Package

**Title** Create Interactive Web Maps with the JavaScript 'Leaflet' Library

**Version** 1.0.0

**Date** 2015-06-19

**Maintainer** Joe Cheng <joe@rstudio.com>

**Description** Create and customize interactive maps using the 'Leaflet' JavaScript library and the 'htmlwidgets' package. These maps can be used directly from the R console, from 'RStudio', in Shiny apps and R Markdown documents.

**License** GPL-3 | file LICENSE

**URL** http://rstudio.github.io/leaflet/

**BugReports** https://github.com/rstudio/leaflet/issues

**Imports** base64enc, htmlwidgets, htmltools, magrittr, markdown, png, RColorBrewer, raster, scales (>= 0.2.5), sp

**Suggests** knitr, maps, shiny, testit (>= 0.4), rgdal, R6, RJSONIO

**NeedsCompilation** no

**Author** Joe Cheng [aut, cre],
Yihui Xie [aut],
Hadley Wickham [ctb],
jQuery Foundation and contributors [ctb, cph] (jQuery library),
Vladimir Agafonkin [ctb, cph] (Leaflet library),
CloudMade [cph] (Leaflet library),
Leaflet contributors [ctb] (Leaflet library),
Leaflet Providers contributors [ctb, cph] (Leaflet Providers plugin),
RStudio [cph]

**Repository** CRAN

**Date/Publication** 2015-06-24 00:56:10

# R topics documented:



---

`addControl` *Graphics elements and layers*

---

## Description

Add graphics elements and layers to the map widget.

## Usage

```
addControl(map, html, position = c("topleft", "topright", "bottomleft", "bottomright"),
    layerId = NULL, className = "info legend", data = getMapData(map))

addTiles(map, urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
    attribution = NULL, layerId = NULL, group = NULL, options = tileOptions())

addWMSTiles(map, baseUrl, layerId = NULL, group = NULL, options = WMSTileOptions(),
    attribution = NULL, layers = "")

addPopups(map, lng = NULL, lat = NULL, popup, layerId = NULL, group = NULL,
    options = popupOptions(), data = getMapData(map))

addMarkers(map, lng = NULL, lat = NULL, layerId = NULL, group = NULL, icon = NULL,
```

```
    popup = NULL, options = markerOptions(), clusterOptions = NULL, clusterId = NULL,
    data = getMapData(map))

addCircleMarkers(map, lng = NULL, lat = NULL, radius = 10, layerId = NULL,
    group = NULL, stroke = TRUE, color = "#03F", weight = 5, opacity = 0.5,
    fill = TRUE, fillColor = color, fillOpacity = 0.2, dashArray = NULL,
    popup = NULL, options = pathOptions(), clusterOptions = NULL, clusterId = NULL,
    data = getMapData(map))

addCircles(map, lng = NULL, lat = NULL, radius = 10, layerId = NULL, group = NULL,
    stroke = TRUE, color = "#03F", weight = 5, opacity = 0.5, fill = TRUE,
    fillColor = color, fillOpacity = 0.2, dashArray = NULL, popup = NULL,
    options = pathOptions(), data = getMapData(map))

addPolylines(map, lng = NULL, lat = NULL, layerId = NULL, group = NULL, stroke = TRUE,
    color = "#03F", weight = 5, opacity = 0.5, fill = FALSE, fillColor = color,
    fillOpacity = 0.2, dashArray = NULL, smoothFactor = 1, noClip = FALSE,
    popup = NULL, options = pathOptions(), data = getMapData(map))

addRectangles(map, lng1, lat1, lng2, lat2, layerId = NULL, group = NULL,
    stroke = TRUE, color = "#03F", weight = 5, opacity = 0.5, fill = TRUE,
    fillColor = color, fillOpacity = 0.2, dashArray = NULL, smoothFactor = 1,
    noClip = FALSE, popup = NULL, options = pathOptions(), data = getMapData(map))

addPolygons(map, lng = NULL, lat = NULL, layerId = NULL, group = NULL, stroke = TRUE,
    color = "#03F", weight = 5, opacity = 0.5, fill = TRUE, fillColor = color,
    fillOpacity = 0.2, dashArray = NULL, smoothFactor = 1, noClip = FALSE,
    popup = NULL, options = pathOptions(), data = getMapData(map))

addGeoJSON(map, geojson, layerId = NULL, group = NULL, stroke = TRUE, color = "#03F",
    weight = 5, opacity = 0.5, fill = TRUE, fillColor = color, fillOpacity = 0.2,
    dashArray = NULL, smoothFactor = 1, noClip = FALSE, options = pathOptions())

addTopoJSON(map, topojson, layerId = NULL, group = NULL, stroke = TRUE, color = "#03F",
    weight = 5, opacity = 0.5, fill = TRUE, fillColor = color, fillOpacity = 0.2,
    dashArray = NULL, smoothFactor = 1, noClip = FALSE, options = pathOptions())
```

## Arguments

| | |
|---|---|
| map | a map widget object created from leaflet() |
| html | the content of the control. May be provided as string or as HTML generated with Shiny/htmltools tags |
| position | position of control: 'topleft', 'topright', 'bottomleft', or 'bottomright' |
| layerId | the layer id |
| className | extra CSS classes to append to the control, space separated |
| data | the data object from which the argument values are derived; by default, it is the data object provided to leaflet() initially, but can be overridden |

| | |
|---|---|
| urlTemplate | a character string as the URL template |
| attribution | the attribution text of the tile layer (HTML) |
| group | the name of the group the newly created layers should belong to (for clearGroup and addLayersControl purposes). Human-friendly group names are permitted–they need not be short, identifier-style names. Any number of layers and even different types of layers (e.g. markers and polygons) can share the same group name. |
| options | a list of extra options for tile layers, popups, paths (circles, rectangles, polygons, ...), or other map elements |
| baseUrl | a base URL of the WMS service |
| layers | comma-separated list of WMS layers to show |
| lng | a numeric vector of longitudes, or a one-sided formula of the form ~x where x is a variable in data; by default (if not explicitly provided), it will be automatically inferred from data by looking for a column named lng, long, or longitude (case-insensitively) |
| lat | a vector of latitudes or a formula (similar to the lng argument; the names lat and latitude are used when guessing the latitude column from data) |
| popup | a character vector of the HTML content for the popups (you are recommended to escape the text using htmlEscape() for security reasons) |
| icon | the icon(s) for markers; an icon is represented by an R list of the form list(iconUrl = '?', iconSize = and you can use icons() to create multiple icons; note when you use an R list that contains images as local files, these local image files will be base64 encoded into the HTML page so the icon images will still be available even when you publish the map elsewhere |
| clusterOptions | if not NULL, markers will be clustered using Leaflet.markercluster; you can use markerClusterOptions() to specify marker cluster options |
| clusterId | the id for the marker cluster layer |
| radius | a numeric vector of radii for the circles; it can also be a one-sided formula, in which case the radius values are derived from the data (units in meters for circles, and pixels for circle markers) |
| stroke | whether to draw stroke along the path (e.g. the borders of polygons or circles) |
| color | stroke color |
| weight | stroke width in pixels |
| opacity | stroke opacity (or layer opacity for tile layers) |
| fill | whether to fill the path with color (e.g. filling on polygons or circles) |
| fillColor | fill color |
| fillOpacity | fill opacity |
| dashArray | a string that defines the stroke dash pattern |
| smoothFactor | how much to simplify the polyline on each zoom level (more means better performance and less accurate representation) |
| noClip | whether to disable polyline clipping |

`lng1, lat1, lng2, lat2`
: latitudes and longitudes of the south-west and north-east corners of rectangles

`geojson`
: a GeoJSON list, or character vector of length 1

`topojson`
: a TopoJSON list, or character vector of length 1

## Value

the new `map` object

## Functions

- `addControl`: Add arbitrary HTML controls to the map
- `addTiles`: Add a tile layer to the map
- `addWMSTiles`: Add a WMS tile layer to the map
- `addPopups`: Add popups to the map
- `addMarkers`: Add markders to the map
- `addCircleMarkers`: Add circle markers to the map
- `addCircles`: Add circles to the map
- `addPolylines`: Add polylines to the map
- `addRectangles`: Add rectangles to the map
- `addPolygons`: Add polygons to the map
- `addGeoJSON`: Add GeoJSON layers to the map
- `addTopoJSON`: Add TopoJSON layers to the map

## References

The Leaflet API documentation: `http://leafletjs.com/reference.html`

## See Also

`tileOptions`, `WMSTileOptions`, `popupOptions`, `markerOptions`, `pathOptions`

---

`addLayersControl` *Add UI controls to switch layers on and off*

---

## Description

Uses Leaflet's built-in layers control feature to allow users to choose one of several base layers, and to choose any number of overlay layers to view.

## Usage

```
addLayersControl(map, baseGroups = character(0), overlayGroups = character(0),
    position = c("topright", "bottomright", "bottomleft", "topleft"),
    options = layersControlOptions())

layersControlOptions(collapsed = TRUE, autoZIndex = TRUE)

removeLayersControl(map)
```

## Arguments

| | |
|---|---|
| map | the map to add the layers control to |
| baseGroups | character vector where each element is the name of a group. The user will be able to choose one base group (only) at a time. This is most commonly used for mostly-opaque tile layers. |
| overlayGroups | character vector where each element is the name of a group. The user can turn each overlay group on or off independently. |
| position | position of control: 'topleft', 'topright', 'bottomleft', or 'bottomright' |
| options | a list of additional options, intended to be provided by a call to `layersControlOptions` |
| collapsed | if `TRUE` (the default), the layers control will be rendered as an icon that expands when hovered over. Set to `FALSE` to have the layers control always appear in its expanded state. |
| autoZIndex | if `TRUE`, the control will automatically maintain the z-order of its various groups as overlays are switched on and off. |

## Examples

```
leaflet() %>% addTiles(group = "OpenStreetMap") %>% addProviderTiles("Stamen.Toner",
    group = "Toner by Stamen") %>% addMarkers(runif(20, -75, -74), runif(20,
    41, 42), group = "Markers") %>% addLayersControl(baseGroups = c("OpenStreetMap",
    "Toner by Stamen"), overlayGroups = c("Markers"))
```

---

# addLegend *Add a color legend to a map*

---

## Description

When a color palette function is used in a map (e.g. `colorNumeric`), a color legend can be automatically derived from the palette function. You can also manually specify the colors and labels for the legend.

## Usage

```
addLegend(map, position = c("topright", "bottomright", "bottomleft", "topleft"),
    pal, values, na.label = "NA", bins = 7, colors, opacity = 0.5, labels,
   labFormat = labelFormat(), title = NULL, className = "info legend", layerId = NULL)

labelFormat(prefix = "", suffix = "", between = " &ndash; ", digits = 3, big.mark = ",",
    transform = identity)
```

## Arguments

| | |
|---|---|
| map | a map widget object created from leaflet() |
| position | the position of the legend |
| pal | the color palette function, generated from colorNumeric(), colorBin(), colorQuantile(), or colorFactor() |
| values | the values used to generate colors from the palette function |
| na.label | the legend label for NAs in values |
| bins | an approximate number of tick-marks on the color gradient for the colorNumeric palette if it is of length one; you can also provide a numeric vector as the predefined breaks (equally spaced) |
| colors | a vector of (HTML) colors to be used in the legend if pal is not provided |
| opacity | the opacity of colors |
| labels | a vector of text labels in the legend corresponding to colors |
| labFormat | a function to format the labels derived from pal and values (see Details below to know what labelFormat() returns by default; you can either use the helper function labelFormat(), or write your own function) |
| title | the legend title |
| className | extra CSS classes to append to the control, space separated |
| layerId | the ID of the legend; subsequent calls to addLegend or addControl with the same layerId will replace this legend. The ID can also be used with removeControl. |
| prefix | a prefix of legend labels |
| suffix | a suffix of legend labels |
| between | a separator between x[i] and x[i + 1] in legend labels (by default, it is a dash) |
| digits | the number of digits of numeric values in labels |
| big.mark | the thousand separator |
| transform | a function to transform the label value |

## Details

The labFormat argument is a function that takes the argument type = c("numeric", "bin", "quantile", "factor"), plus, arguments for different types of color palettes. For the colorNumeric() palette, labFormat takes a single argument, which is the breaks of the numeric vector, and returns a character vector of the same length. For colorBin(), labFormat also takes a vector of breaks of length n but

should return a character vector of length `n - 1`, with the `i`-th element representing the interval `c(x[i], x[i + 1])`. For `colorQuantile`, `labFormat` takes two arguments, the quantiles and the associated probabilities (each of length `n`), and should return a character vector of length `n - 1` (similar to the `colorBin()` palette). For `colorFactor()`, `labFormat` takes one argument, the unique values of the factor, and should return a character vector of the same length.

By default, `labFormat` is basically `format(scientific = FALSE,big.mark = ',')` for the numeric palette, `as.character()` for the factor palette, and a function to return labels of the form '`x[i] - x[i + 1]`' for bin and quantile palettes (in the case of quantile palettes, `x` is the probabilities instead of the values of breaks).

## Examples

```
library(leaflet)
# a manual legend
leaflet() %>% addTiles() %>% addLegend(
  position = 'bottomright',
  colors = rgb(t(col2rgb(palette())) / 255),
  labels = palette(), opacity = 1,
  title = 'An Obvious Legend'
)

# an automatic legend derived from the color palette
df = local({
  n = 300; x = rnorm(n); y = rnorm(n)
  z = sqrt(x^2 + y^2); z[sample(n, 10)] = NA
  data.frame(x, y, z)
})
pal = colorNumeric('OrRd', df$z)
leaflet(df) %>%
  addCircleMarkers(~x, ~y, color = ~pal(z)) %>%
  addLegend(pal = pal, values = ~z)

# format legend labels
df = data.frame(x = rnorm(100), y = rexp(100, 2), z = runif(100))
pal = colorBin('PuOr', df$z, bins = c(0, .1, .4, .9, 1))
leaflet(df) %>%
  addCircleMarkers(~x, ~y, color = ~pal(z)) %>%
  addLegend(pal = pal, values = ~z)

leaflet(df) %>%
  addCircleMarkers(~x, ~y, color = ~pal(z)) %>%
  addLegend(pal = pal, values = ~z, labFormat = labelFormat(
    prefix = '(', suffix = ')%', between = ', ',
    transform = function(x) 100 * x
  ))
```

---

`addProviderTiles` *Add a tile layer from a known map provider*

---

## Description

Add a tile layer from a known map provider

## Usage

```
addProviderTiles(map, provider, layerId = NULL, group = NULL,
    options = providerTileOptions())

providerTileOptions(errorTileUrl = "", noWrap = FALSE, opacity = NULL,
    zIndex = NULL, unloadInvisibleTiles = NULL, updateWhenIdle = NULL,
    detectRetina = FALSE, reuseTiles = FALSE, ...)
```

## Arguments

| | |
|---|---|
| map | the map to add the tile layer to |
| provider | the name of the provider (see http://leaflet-extras.github.io/leaflet-providers/preview/ and https://github.com/leaflet-extras/leaflet-providers) |
| layerId | the layer id to assign |
| group | the name of the group the newly created layers should belong to (for clearGroup and addLayersControl purposes). Human-friendly group names are permitted–they need not be short, identifier-style names. |
| options | tile options |
| errorTileUrl, noWrap, opacity, zIndex, unloadInvisibleTiles, updateWhenIdle, detectRetina, reuseTil | the tile layer options; see http://leafletjs.com/reference.html#tilelayer |
| ... | named parameters to add to the options |

## Value

modified map object

## Examples

```
leaflet() %>% addProviderTiles("Stamen.Watercolor") %>%
    addProviderTiles("Stamen.TonerHybrid")
```

---

# addRasterImage *Add a raster image as a layer*

---

## Description

Create an image overlay from a RasterLayer object. *This is only suitable for small to medium sized rasters*, as the entire image will be embedded into the HTML page (or passed over the websocket in a Shiny context).

## Usage

```
addRasterImage(map, x, colors = "Spectral", opacity = 1, attribution = NULL,
    layerId = NULL, group = NULL, project = TRUE, maxBytes = 4 * 1024 * 1024)

projectRasterForLeaflet(x)
```

## Arguments

| | |
|---|---|
| map | a map widget object |
| x | a RasterLayer object–see raster |
| colors | the color palette (see colorNumeric) or function to use to color the raster values (hint: if providing a function, set na.color to "#00000000" to make NA areas transparent) |
| opacity | the base opacity of the raster, expressed from 0 to 1 |
| attribution | the HTML string to show as the attribution for this layer |
| layerId | the layer id |
| group | the name of the group this raster image should belong to (see the same parameter under addTiles) |
| project | if TRUE, automatically project x to the map projection expected by Leaflet (EPSG:3857); if FALSE, it's the caller's responsibility to ensure that x is already projected, and that extent(x) is expressed in WGS84 latitude/longitude coordinates |
| maxBytes | the maximum number of bytes to allow for the projected image (before base64 encoding); defaults to 4MB. |

## Details

The maxBytes parameter serves to prevent you from accidentally embedding an excessively large amount of data into your htmlwidget. This value is compared to the size of the final compressed image (after the raster has been projected, colored, and PNG encoded, but before base64 encoding is applied). Set maxBytes to Inf to disable this check, but be aware that very large rasters may not only make your map a large download but also may cause the browser to become slow or unresponsive.

By default, the addRasterImage function will project the RasterLayer x to EPSG:3857 using the raster package's projectRaster function. This can be a time-consuming operation for even moderately sized rasters. Upgrading the raster package to 2.4 or later will provide a large speedup versus previous versions. If you are repeatedly adding a particular raster to your Leaflet maps, you can perform the projection ahead of time using projectRasterForLeaflet(), and call addRasterImage with project=FALSE.

## Examples

```
library(raster)

r <- raster(xmn = -2.8, xmx = -2.79, ymn = 54.04, ymx = 54.05, nrows = 30, ncols = 30)
values(r) <- matrix(1:900, nrow(r), ncol(r), byrow = TRUE)
crs(r) <- CRS("+init=epsg:4326")
```

```
leaflet() %>% addTiles() %>% addRasterImage(r, colors = "Spectral", opacity = 0.8)
```

---

`colorNumeric` *Color mapping*

---

## Description

Conveniently maps data values (numeric or factor/character) to colors according to a given palette, which can be provided in a variety of formats.

## Usage

```
colorNumeric(palette, domain, na.color = "#808080", alpha = FALSE)

colorBin(palette, domain, bins = 7, pretty = TRUE, na.color = "#808080", alpha = FALSE)

colorQuantile(palette, domain, n = 4, probs = seq(0, 1, length.out = n + 1),
    na.color = "#808080", alpha = FALSE)

colorFactor(palette, domain, levels = NULL, ordered = FALSE, na.color = "#808080",
    alpha = FALSE)
```

## Arguments

| | |
|---|---|
| `palette` | The colors or color function that values will be mapped to |
| `domain` | The possible values that can be mapped.<br>For `colorNumeric` and `colorBin`, this can be a simple numeric range (e.g. `c(0, 100)`); `colorQuantile` needs representative numeric data; and `colorFactor` needs categorical data.<br>If `NULL`, then whenever the resulting color function is called, the `x` value will represent the domain. This implies that if the function is invoked multiple times, the encoding between values and colors may not be consistent; if consistency is needed, you must provide a non-`NULL` domain. |
| `na.color` | The color to return for `NA` values. Note that `na.color=NA` is valid. |
| `alpha` | Whether alpha channels should be respected or ignored. If `TRUE` then colors without explicit alpha information will be treated as fully opaque. |
| `bins` | Either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which the domain values are to be cut. |
| `pretty` | Whether to use the function `pretty()` to generate the bins when the argument `bins` is a single number. When `pretty = TRUE`, the actual number of bins may not be the number of bins you specified. When `pretty = FALSE`, `seq()` is used to generate the bins and the breaks may not be "pretty". |

| | |
|---|---|
| n | Number of equal-size quantiles desired. For more precise control, use the probs argument instead. |
| probs | See quantile. If provided, the n argument is ignored. |
| levels | An alternate way of specifying levels; if specified, domain is ignored |
| ordered | If TRUE and domain needs to be coerced to a factor, treat it as already in the correct order |

## Details

colorNumeric is a simple linear mapping from continuous numeric data to an interpolated palette.

colorBin also maps continuous numeric data, but performs binning based on value (see the cut function).

colorQuantile similarly bins numeric data, but via the quantile function.

colorFactor maps factors to colors. If the palette is discrete and has a different number of colors than the number of factors, interpolation is used.

The palette argument can be any of the following:

1. A character vector of RGB or named colors. Examples: palette(), c("#000000", "#0000FF", "#FFFFFF"), topo.colors(10)
2. The name of an RColorBrewer palette, e.g. "BuPu" or "Greens".
3. A function that receives a single value between 0 and 1 and returns a color. Examples: colorRamp(c("#000000", "#FFFFFF"), interpolate="spline").

## Value

A function that takes a single parameter x; when called with a vector of numbers (except for colorFactor, which expects factors/characters), #RRGGBB color strings are returned (unless alpha=TRUE in which case #RRGGBBAA may also be possible).

## Examples

```
pal = colorBin("Greens", domain = 0:100)
pal(runif(10, 60, 100))


# Exponential distribution, mapped continuously
previewColors(colorNumeric("Blues", domain = NULL), sort(rexp(16)))
# Exponential distribution, mapped by interval
previewColors(colorBin("Blues", domain = NULL, bins = 4), sort(rexp(16)))
# Exponential distribution, mapped by quantile
previewColors(colorQuantile("Blues", domain = NULL), sort(rexp(16)))

# Categorical data; by default, the values being colored span the gamut...
previewColors(colorFactor("RdYlBu", domain = NULL), LETTERS[1:5])
# ...unless the data is a factor, without droplevels...
previewColors(colorFactor("RdYlBu", domain = NULL), factor(LETTERS[1:5],
    levels = LETTERS))
# ...or the domain is stated explicitly.
```

```
previewColors(colorFactor("RdYlBu", levels = LETTERS), LETTERS[1:5])
```

---

`createLeafletMap` *Legacy functions*

---

## Description

These functions are provided for backwards compatibility with the first iteration of the leaflet bindings (https://github.com/jcheng5/leaflet-shiny).

## Usage

```
createLeafletMap(session, outputId)

leafletMap(outputId, width, height,
  initialTileLayer = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
  initialTileLayerAttribution = NULL, options = NULL)
```

## Arguments

`session, outputId`
: Deprecated

`width, height, initialTileLayer, initialTileLayerAttribution, options`
: Deprecated

---

`dispatch` *Extension points for plugins*

---

## Description

Extension points for plugins

## Usage

```
dispatch(map, funcName, leaflet = stop(paste(funcName, "requires a map proxy object")),
    leaflet_proxy = stop(paste(funcName, "does not support map proxy objects")))

invokeMethod(map, data, method, ...)
```

### Arguments

| | |
|---|---|
| map | a map object, as returned from leaflet or leafletProxy |
| funcName | the name of the function that the user called that caused this dispatch call; for error message purposes |
| leaflet | an action to be performed if the map is from leaflet |
| leaflet_proxy | an action to be performed if the map is from leafletProxy |
| data | a data object that will be used when evaluating formulas in ... |
| method | the name of the JavaScript method to invoke |
| ... | unnamed arguments to be passed to the JavaScript method |

### Value

dispatch returns the value of leaflet or leaflet_proxy, or an error. invokeMethod returns the map object that was passed in, possibly modified.

---

## iconList *Make icon set*

---

### Description

Make icon set

### Usage

```
iconList(...)
```

### Arguments

| | |
|---|---|
| ... | icons created from makeIcon() |

### Examples

```
iconSet = iconList(red = makeIcon("leaf-red.png", iconWidth = 32, iconHeight = 32),
    green = makeIcon("leaf-green.png", iconWidth = 32, iconHeight = 32))

iconSet[c("red", "green", "red")]
```

icons *Create a list of icon data*

## Description

An icon can be represented as a list of the form `list(iconUrl,iconSize, ...)`. This function is vectorized over its arguments to create a list of icon data. Shorter argument values will be re-cycled. `NULL` values for these arguments will be ignored.

## Usage

```
icons(iconUrl = NULL, iconRetinaUrl = NULL, iconWidth = NULL, iconHeight = NULL,
   iconAnchorX = NULL, iconAnchorY = NULL, shadowUrl = NULL, shadowRetinaUrl = NULL,
   shadowWidth = NULL, shadowHeight = NULL, shadowAnchorX = NULL, shadowAnchorY = NULL,
    popupAnchorX = NULL, popupAnchorY = NULL, className = NULL)
```

## Arguments

| | |
|---|---|
| `iconUrl` | the URL or file path to the icon image |
| `iconRetinaUrl` | the URL or file path to a retina sized version of the icon image |
| `iconWidth, iconHeight` | size of the icon image in pixels |
| `iconAnchorX, iconAnchorY` | the coordinates of the "tip" of the icon (relative to its top left corner, i.e. the top left corner means `iconAnchorX = 0` and `iconAnchorY = 0`), and the icon will be aligned so that this point is at the marker's geographical location |
| `shadowUrl` | the URL or file path to the icon shadow image |
| `shadowRetinaUrl` | the URL or file path to the retina sized version of the icon shadow image |
| `shadowWidth, shadowHeight` | size of the shadow image in pixels |
| `shadowAnchorX, shadowAnchorY` | the coordinates of the "tip" of the shadow |
| `popupAnchorX, popupAnchorY` | the coordinates of the point from which popups will "open", relative to the icon anchor |
| `className` | a custom class name to assign to both icon and shadow images |

## Value

A list of icon data that can be passed to the `icon` argument of `addMarkers()`.

## Examples

```
library(leaflet)

# adapted from http://leafletjs.com/examples/custom-icons.html

iconData = data.frame(
  lat = c(rnorm(10, 0), rnorm(10, 1), rnorm(10, 2)),
  lng = c(rnorm(10, 0), rnorm(10, 3), rnorm(10, 6)),
  group = rep(sort(c('green', 'red', 'orange')), each = 10),
  stringsAsFactors = FALSE
)

leaflet() %>% addMarkers(
  data = iconData,
  icon = ~ icons(
    iconUrl = sprintf('http://leafletjs.com/docs/images/leaf-%s.png', group),
    shadowUrl = 'http://leafletjs.com/docs/images/leaf-shadow.png',
    iconWidth = 38, iconHeight = 95, shadowWidth = 50, shadowHeight = 64,
    iconAnchorX = 22, iconAnchorY = 94, shadowAnchorX = 4, shadowAnchorY = 62,
    popupAnchorX = -3, popupAnchorY = -76
  )
)


# use point symbols from base R graphics as icons
pchIcons = function(pch = 0:14, width = 30, height = 30, ...) {
  n = length(pch)
  files = character(n)
  # create a sequence of png images
  for (i in seq_len(n)) {
    f = tempfile(fileext = '.png')
    png(f, width = width, height = height, bg = 'transparent')
    par(mar = c(0, 0, 0, 0))
    plot.new()
    points(.5, .5, pch = pch[i], cex = min(width, height) / 8, ...)
    dev.off()
    files[i] = f
  }
  files
}

iconData = matrix(rnorm(500), ncol = 2)
res = kmeans(iconData, 10)
iconData = cbind(iconData, res$cluster)
colnames(iconData) = c('lat', 'lng', 'group')
iconData = as.data.frame(iconData)

# 10 random point shapes for the 10 clusters in iconData
shapes = sample(0:14, 10)
iconFiles = pchIcons(shapes, 40, 40, col = 'steelblue', lwd = 2)

# note the data has 250 rows, and there are 10 icons in iconFiles; they are
```

```
# connected by the `group` variable: the i-th row of iconData uses the
# group[i]-th icon in the icon list
leaflet() %>% addMarkers(
  data = iconData,
  icon = ~ icons(
    iconUrl = iconFiles[group],
    popupAnchorX = 20, popupAnchorY = 0
  ),
  popup = ~ sprintf(
    'lat = %.4f, long = %.4f, group = %s, pch = %s', lat, lng, group, shapes[group]
  )
)

unlink(iconFiles)  # clean up the tmp png files that have been embedded
```

---

`leaflet` *Create a Leaflet map widget*

---

## Description

This function creates a Leaflet map widget using **htmlwidgets**. The widget can be rendered on HTML pages generated from R Markdown, Shiny, or other applications.

## Usage

```
leaflet(data = NULL, width = NULL, height = NULL, padding = 0)
```

## Arguments

| | |
|---|---|
| `data` | a data object (currently supported objects are matrices, data frames, and spatial objects from the **sp** package of classes `SpatialPoints`, `SpatialPointsDataFrame`, `Polygon`, `Polygons`, `SpatialPolygons`, `SpatialPolygonsDataFrame`, `Line`, `Lines`, `SpatialLines`, and `SpatialLinesDataFrame`) |
| `width` | the width of the map |
| `height` | the height of the map |
| `padding` | the padding of the map |

## Details

The `data` argument is only needed if you are going to reference variables in this object later in map layers. For example, `data` can be a data frame containing columns `latitude` and `longtitude`, then we may add a circle layer to the map by `leaflet(data) %>% addCircles(lat = ~latitude, lng = ~longtitude)`, where the variables in the formulae will be evaluated in the `data`.

## Value

A HTML widget object, on which we can add graphics layers using `%>%` (see examples).

## Examples

```
library(leaflet)
m = leaflet() %>% addTiles()
m  # a map with the default OSM tile layer

# set bounds
m %>% fitBounds(0, 40, 10, 50)

# move the center to Snedecor Hall
m = m %>% setView(-93.65, 42.0285, zoom = 17)
m

# popup
m %>% addPopups(-93.65, 42.0285, 'Here is the <b>Department of Statistics</b>, ISU')
rand_lng = function(n = 10) rnorm(n, -93.65, .01)
rand_lat = function(n = 10) rnorm(n, 42.0285, .01)

# use automatic bounds derived from lng/lat data
m = m %>% clearBounds()

# popup
m %>% addPopups(rand_lng(), rand_lat(), 'Random popups')

# marker
m %>% addMarkers(rand_lng(), rand_lat())
m %>% addMarkers(
  rand_lng(), rand_lat(), popup = paste('A random letter', sample(LETTERS, 10))
)

Rlogo = file.path(R.home('doc'), 'html', 'logo.jpg')
m %>% addMarkers(
  174.7690922, -36.8523071, icon = list(
    iconUrl = Rlogo, iconSize = c(100, 76)
  ), popup = 'R was born here!'
)

m %>% addMarkers(rnorm(30, 175), rnorm(30, -37), icon = list(
  iconUrl = Rlogo, iconSize = c(25, 19)
))

m %>% addMarkers(
  c(-71.0382679, -122.1217866), c(42.3489054, 47.6763144), icon = list(
    iconUrl = 'http://www.rstudio.com/wp-content/uploads/2014/03/blue-125.png'
  ), popup = c('RStudio @ Boston', 'RStudio @ Seattle')
)


# circle (units in metres)
m %>% addCircles(rand_lng(50), rand_lat(50), radius = runif(50, 50, 150))

# circle marker (units in pixels)
m %>% addCircleMarkers(rand_lng(50), rand_lat(50), color = '#ff0000')
```

```
m %>% addCircleMarkers(rand_lng(100), rand_lat(100), radius = runif(100, 5, 15))

# rectangle
m %>% addRectangles(
  rand_lng(), rand_lat(), rand_lng(), rand_lat(),
  color = 'red', fill = FALSE, dashArray = '5,5', weight = 3
)

# polyline
m %>% addPolylines(rand_lng(50), rand_lat(50))

# polygon
m %>% addPolygons(rand_lng(), rand_lat(), layerId = 'foo')

# geoJSON
seattle_geojson = list(
  type = "Feature",
  geometry = list(
    type = "MultiPolygon",
    coordinates = list(list(list(
      c(-122.36075812146,  47.6759920119894),
      c(-122.360781646764, 47.6668890126755),
      c(-122.360782108665,  47.6614990696722),
      c(-122.366199035722, 47.6614990696722),
      c(-122.366199035722,  47.6592874248973),
      c(-122.364582509469, 47.6576254522105),
      c(-122.363887331445,  47.6569107302038),
      c(-122.360865528129, 47.6538418253251),
      c(-122.360866157644,  47.6535254473167),
      c(-122.360866581103, 47.6533126275176),
      c(-122.362526540691,  47.6541872926348),
      c(-122.364442114483, 47.6551892850798),
      c(-122.366077719797,  47.6560733960606),
      c(-122.368818463838, 47.6579742346694),
      c(-122.370115159943,  47.6588730808334),
      c(-122.372295967029, 47.6604350102328),
      c(-122.37381369088,  47.660582362063),
      c(-122.375522972109, 47.6606413027949),
      c(-122.376079703095,  47.6608793094619),
      c(-122.376206315662, 47.6609242364243),
      c(-122.377610811371,  47.6606160735197),
      c(-122.379857378879, 47.6610306942278),
      c(-122.382454873022,  47.6627496239169),
      c(-122.385357955057, 47.6638573778241),
      c(-122.386007328104,  47.6640865692306),
      c(-122.387186331506, 47.6654326177161),
      c(-122.387802656231,  47.6661492860294),
      c(-122.388108244121, 47.6664548739202),
      c(-122.389177800763,  47.6663784774359),
      c(-122.390582858689, 47.6665072251861),
      c(-122.390793942299,  47.6659699214511),
      c(-122.391507906234, 47.6659200946229),
      c(-122.392883050767,  47.6664166747017),
```

```
      c(-122.392847210144, 47.6678696739431),
      c(-122.392904778401,  47.6709016021624),
      c(-122.39296705153, 47.6732047491624),
      c(-122.393000803496,  47.6759322346303),
      c(-122.37666945305, 47.6759896300663),
      c(-122.376486363943,  47.6759891899754),
      c(-122.366078869215, 47.6759641734893),
      c(-122.36075812146,  47.6759920119894)
    )))
  ),
  properties = list(
    name = "Ballard",
    population = 48000,
    # You can inline styles if you want
    style = list(
      fillColor = "yellow",
      weight = 2,
      color = "#000000"
    )
  ),
  id = "ballard"
)
m %>% setView(-122.36075812146, 47.6759920119894, zoom = 13) %>% addGeoJSON(seattle_geojson)


# use the Dark Matter layer from CartoDB
leaflet() %>% addTiles('http://{s}.basemaps.cartocdn.com/dark_all/{z}/{x}/{y}.png',
  attribution = paste(
    '&copy; <a href="http://openstreetmap.org">OpenStreetMap</a> contributors',
    '&copy; <a href="http://cartodb.com/attributions">CartoDB</a>'
  )
) %>% setView(-122.36, 47.67, zoom = 10)

# provide a data frame to leaflet()
categories = LETTERS[1:10]
df = data.frame(
  lat = rand_lat(100), lng = rand_lng(100), size = runif(100, 5, 20),
  category = factor(sample(categories, 100, replace = TRUE), levels = categories),
  value = rnorm(100)
)
m = leaflet(df) %>% addTiles()
m %>% addCircleMarkers(~lng, ~lat, radius = ~size)
m %>% addCircleMarkers(~lng, ~lat, radius = runif(100, 4, 10), color = c('red'))

# Discrete colors using the "RdYlBu" colorbrewer palette, mapped to categories
RdYlBu = colorFactor("RdYlBu", domain = categories)
m %>% addCircleMarkers(~lng, ~lat, radius = ~size,
  color = ~RdYlBu(category), fillOpacity = 0.5)

# Continuous colors using the "Greens" colorbrewer palette, mapped to value
greens = colorNumeric("Greens", domain = NULL)
m %>% addCircleMarkers(~lng, ~lat, radius = ~size,
  color = ~greens(value), fillOpacity = 0.5)
```

## leaflet-imports *Objects imported from other packages*

### Description

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

**htmlwidgets** JS

**magrittr** %>%

## leafletOutput *Wrapper functions for using* **leaflet** *in* **shiny**

### Description

Use `leafletOutput()` to create a UI element, and `renderLeaflet()` to render the map widget.

### Usage

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

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

### Arguments

| | |
|---|---|
| `outputId` | output variable to read from |
| `width, height` | the width and height of the map (see `shinyWidgetOutput`) |
| `expr` | An expression that generates an HTML widget |
| `env` | The environment in which to evaluate `expr`. |
| `quoted` | Is `expr` a quoted expression (with `quote()`)? This is useful if you want to save an expression in a variable. |

### Examples

```
library(leaflet)
library(shiny)
app = shinyApp(
  ui = fluidPage(leafletOutput('myMap')),
  server = function(input, output) {
    map = leaflet() %>% addTiles() %>% setView(-93.65, 42.0285, zoom = 17)
    output$myMap = renderLeaflet(map)
  }
)

if (interactive()) print(app)
```

`leafletProxy` *Send commands to a Leaflet instance in a Shiny app*

## Description

Creates a map-like object that can be used to customize and control a map that has already been rendered. For use in Shiny apps and Shiny docs only.

## Usage

```
leafletProxy(mapId, session = shiny::getDefaultReactiveDomain(), data = NULL,
    deferUntilFlush = TRUE)
```

## Arguments

| | |
|---|---|
| `mapId` | single-element character vector indicating the output ID of the map to modify |
| `session` | the Shiny session object to which the map belongs; usually the default value will suffice |
| `data` | a data object; see Details under the `leaflet` help topic |
| `deferUntilFlush` | indicates whether actions performed against this instance should be carried out right away, or whether they should be held until after the next time all of the outputs are updated; defaults to `TRUE` |

## Details

Normally, you create a Leaflet map using the `leaflet` function. This creates an in-memory representation of a map that you can customize using functions like `addPolygons` and `setView`. Such a map can be printed at the R console, included in an R Markdown document, or rendered as a Shiny output.

In the case of Shiny, you may want to further customize a map, even after it is rendered to an output. At this point, the in-memory representation of the map is long gone, and the user's web browser has already realized the Leaflet map instance.

This is where `leafletProxy` comes in. It returns an object that can stand in for the usual Leaflet map object. The usual map functions like `addPolygons` and `setView` can be called, and instead of customizing an in-memory representation, these commands will execute on the live Leaflet map instance.

## Examples

```
library(shiny)

ui <- fluidPage(leafletOutput("map1"))

server <- function(input, output, session) {
```

```
    output$map1 <- renderLeaflet({
        leaflet() %>% addCircleMarkers(lng = runif(10), lat = runif(10),
            layerId = paste0("marker", 1:10))
    })

    observeEvent(input$map1_marker_click, {
        leafletProxy("map1", session) %>% removeMarker(input$map1_marker_click$id)
    })
}

shinyApp(ui, server)
```

---

`makeIcon` *Define icon sets*

---

## Description

Define icon sets

## Usage

```
makeIcon(iconUrl = NULL, iconRetinaUrl = NULL, iconWidth = NULL, iconHeight = NULL,
   iconAnchorX = NULL, iconAnchorY = NULL, shadowUrl = NULL, shadowRetinaUrl = NULL,
   shadowWidth = NULL, shadowHeight = NULL, shadowAnchorX = NULL, shadowAnchorY = NULL,
    popupAnchorX = NULL, popupAnchorY = NULL, className = NULL)
```

## Arguments

| | |
|---|---|
| `iconUrl` | the URL or file path to the icon image |
| `iconRetinaUrl` | the URL or file path to a retina sized version of the icon image |
| `iconWidth` | size of the icon image in pixels |
| `iconHeight` | size of the icon image in pixels |
| `iconAnchorX` | the coordinates of the "tip" of the icon (relative to its top left corner, i.e. the top left corner means `iconAnchorX = 0` and `iconAnchorY = 0`), and the icon will be aligned so that this point is at the marker's geographical location |
| `iconAnchorY` | the coordinates of the "tip" of the icon (relative to its top left corner, i.e. the top left corner means `iconAnchorX = 0` and `iconAnchorY = 0`), and the icon will be aligned so that this point is at the marker's geographical location |
| `shadowUrl` | the URL or file path to the icon shadow image |
| `shadowRetinaUrl` | the URL or file path to the retina sized version of the icon shadow image |
| `shadowWidth` | size of the shadow image in pixels |
| `shadowHeight` | size of the shadow image in pixels |

| | |
|---|---|
| shadowAnchorX | the coordinates of the "tip" of the shadow |
| shadowAnchorY | the coordinates of the "tip" of the shadow |
| popupAnchorX | the coordinates of the point from which popups will "open", relative to the icon anchor |
| popupAnchorY | the coordinates of the point from which popups will "open", relative to the icon anchor |
| className | a custom class name to assign to both icon and shadow images |

---

`mapOptions` *Set options on a leaflet map object*

---

## Description

Set options on a leaflet map object

## Usage

```
mapOptions(map, zoomToLimits = c("always", "first", "never"))
```

## Arguments

| | |
|---|---|
| map | A map widget object created from `leaflet()` |
| zoomToLimits | Controls whether the map is zooms to the limits of the elements on the map. This is useful for interactive applications where the map data is updated. If `"always"` (the default), the map always re-zooms when new data is received; if `"first"`, it zooms to the elements on the first rendering, but does not re-zoom for subsequent data; if `"never"`, it never re-zooms, not even for the first rendering. |

## Examples

```
# Don't auto-zoom to the objects (can be useful in interactive applications)
leaflet() %>% addTiles() %>% addPopups(174.7690922, -36.8523071, "R was born here!") %>%
    mapOptions(zoomToLimits = "first")
```

## previewColors *Color previewing utility*

### Description

Color previewing utility

### Usage

```
previewColors(pal, values)
```

### Arguments

| | |
|---|---|
| pal | A color mapping function, like those returned from colorNumeric, et al |
| values | A set of values to preview colors for |

### Value

An HTML-based list of the colors and values

## removeControl *Remove elements from a map*

### Description

Remove one or more features from a map, identified by layerId; or, clear all features of the given type or group.

### Usage

```
removeControl(map, layerId)

clearControls(map)

clearGroup(map, group)

removeImage(map, layerId)

clearImages(map)

removeTiles(map, layerId)

clearTiles(map)

removePopup(map, layerId)
```

```
clearPopups(map)

removeMarker(map, layerId)

clearMarkers(map)

removeMarkerCluster(map, layerId)

clearMarkerClusters(map)

removeMarkerFromCluster(map, layerId, clusterId)

removeShape(map, layerId)

clearShapes(map)

removeGeoJSON(map, layerId)

clearGeoJSON(map)
```

## Arguments

| | |
|---|---|
| map | a map widget object, possibly created from leaflet() but more likely from leafletProxy() |
| layerId | character vector; the layer id(s) of the item to remove |
| group | the name of the group whose members should be removed |
| clusterId | the id of the marker cluster layer |

## Value

the new map object

## Note

When used with a leaflet() map object, these functions don't actually remove the features from the map object, but simply add an operation that will cause those features to be removed after they are added. In other words, if you add a polygon "foo" and the call removeShape("foo"), it's not smart enough to prevent the polygon from being added in the first place; instead, when the map is rendered, the polygon will be added and then removed.

For that reason, these functions aren't that useful with leaflet map objects and are really intended to be used with leafletProxy instead.

WMS tile layers are extensions of tile layers, so they can also be removed or cleared via removeTiles() or clearTiles().

setView *Methods to manipulate the map widget*

## Description

A series of methods to manipulate the map.

## Usage

```
setView(map, lng, lat, zoom, options = list())

fitBounds(map, lng1, lat1, lng2, lat2)

setMaxBounds(map, lng1, lat1, lng2, lat2)

clearBounds(map)
```

## Arguments

| | |
|---|---|
| map | a map widget object created from leaflet() |
| lng | The longitude of the map center |
| lat | The latitude of the map center |
| zoom | the zoom level |
| options | a list of zoom/pan options (see http://leafletjs.com/reference.html#map-zoompanoptions) |
| lng1, lat1, lng2, lat2 | the coordinates of the map bounds |

## Value

The modified map widget.

## Functions

- setView: Set the view of the map (center and zoom level)
- fitBounds: Set the bounds of a map
- setMaxBounds: Restricts the map view to the given bounds
- clearBounds: Clear the bounds of a map, and the bounds will be automatically determined from latitudes and longitudes of the map elements if available (otherwise the full world view is used)

## References

http://leafletjs.com/reference.html#map-set-methods

## Examples

```
library(leaflet)
m = leaflet() %>% addTiles() %>% setView(-71.0382679, 42.3489054, zoom = 18)
m  # the RStudio 'headquarter'
m %>% fitBounds(-72, 40, -70, 43)
m %>% clearBounds()  # world view
```

---

showGroup *Show or hide layer groups*

---

## Description

Hide groups of layers without removing them from the map entirely. Groups are created using the group parameter that is included on most layer adding functions.

## Usage

```
showGroup(map, group)

hideGroup(map, group)
```

## Arguments

| | |
|---|---|
| map | the map to modify |
| group | character vector of one or more group names to show or hide |

## See Also

addLayersControl to allow users to show/hide layer groups interactively

---

tileOptions *Extra options for map elements and layers*

---

## Description

The rest of all possible options for map elements and layers that are not listed in the layer functions.

## Usage

```
tileOptions(minZoom = 0, maxZoom = 18, maxNativeZoom = NULL, tileSize = 256,
    subdomains = "abc", errorTileUrl = "", tms = FALSE, continuousWorld = FALSE,
    noWrap = FALSE, zoomOffset = 0, zoomReverse = FALSE, opacity = 1, zIndex = NULL,
    unloadInvisibleTiles = NULL, updateWhenIdle = NULL, detectRetina = FALSE,
    reuseTiles = FALSE)

WMSTileOptions(styles = "", format = "image/jpeg", transparent = FALSE,
    version = "1.1.1", crs = NULL, ...)

popupOptions(maxWidth = 300, minWidth = 50, maxHeight = NULL, autoPan = TRUE,
    keepInView = FALSE, closeButton = TRUE, zoomAnimation = TRUE, closeOnClick = NULL,
    className = "")

markerOptions(clickable = TRUE, draggable = FALSE, keyboard = TRUE, title = "",
    alt = "", zIndexOffset = 0, opacity = 1, riseOnHover = FALSE, riseOffset = 250)

markerClusterOptions(showCoverageOnHover = TRUE, zoomToBoundsOnClick = TRUE,
    spiderfyOnMaxZoom = TRUE, removeOutsideVisibleBounds = TRUE, ...)

pathOptions(lineCap = NULL, lineJoin = NULL, clickable = TRUE, pointerEvents = NULL,
    className = "")
```

## Arguments

`minZoom, maxZoom, maxNativeZoom, tileSize, subdomains, errorTileUrl, tms, continuousWorld, noWrap,`
: the tile layer options; see http://leafletjs.com/reference.html#tilelayer

`styles`
: comma-separated list of WMS styles

`format`
: WMS image format (use `'image/png'` for layers with transparency)

`transparent`
: if `TRUE`, the WMS service will return images with transparency

`version`
: version of the WMS service to use

`crs`
: Coordinate Reference System to use for the WMS requests, defaults to map CRS (don't change this if you're not sure what it means)

`...`
: other tile options for `WMSTileOptions()` (all arguments of `tileOptions()` can be used)

`maxWidth, minWidth, maxHeight, autoPan, keepInView, closeButton, zoomAnimation, closeOnClick`
: popup options; see http://leafletjs.com/reference.html#popup

`className`
: a CSS class name set on an element

`clickable`
: whether the element emits mouse events

`draggable, keyboard, title, alt, zIndexOffset, opacity, riseOnHover, riseOffset`
: marker options; see http://leafletjs.com/reference.html#marker

`showCoverageOnHover`
: when you mouse over a cluster it shows the bounds of its markers

`zoomToBoundsOnClick`
: when you click a cluster we zoom to its bounds

`spiderfyOnMaxZoom`
: when you click a cluster at the bottom zoom level we spiderfy it so you can see all of its markers

`removeOutsideVisibleBounds`
: clusters and markers too far from the viewport are removed from the map for performance

`lineCap`
: a string that defines shape to be used at the end of the stroke

`lineJoin`
: a string that defines shape to be used at the corners of the stroke

`pointerEvents`
: sets the `pointer-events` attribute on the path if SVG backend is used

## Functions

- `tileOptions`: Options for tile layers
- `WMSTileOptions`: Options for WMS tile layers
- `popupOptions`: Options for popups
- `markerOptions`: Options for markers
- `markerClusterOptions`: Options for marker clusters
- `pathOptions`: Options for vector layers (polylines, polygons, rectangles, and circles, etc)

# Index