Package 'googleway'

July 22, 2025

Type Package

Title Accesses Google Maps APIs to Retrieve Data and Plot Maps

Version 2.7.8

Date 2023-08-22

Description Provides a mechanism to plot a 'Google Map' from 'R' and overlay it with shapes and markers. Also provides access to 'Google Maps' APIs, including places, directions, roads, distances, geocoding, elevation and timezone.

License MIT + file LICENSE

LazyData TRUE

Depends R (>= 2.10.0)

Imports jsonlite (>= 0.9.20), curl, htmlwidgets, htmltools, magrittr, shiny, jpeg, utils, jqr, viridisLite, scales, grDevices, googlePolylines (>= 0.7.1)

RoxygenNote 7.2.3

BugReports https://github.com/SymbolixAU/googleway/issues

Suggests knitr, rmarkdown, testthat

VignetteBuilder knitr

Encoding UTF-8

NeedsCompilation no

Author David Cooley [aut, cre],
Paulo Barcelos [ctb] (Author of c++ decode_pl),
Rstudio [ctb] (Functions written for the Leaflet pacakge)

Maintainer David Cooley <dcooley@symbolix.com.au>

Repository CRAN

Date/Publication 2023-08-22 06:40:06 UTC

Contents

`access_result` *Access Result*

Description

Methods for accessing specific elements of a Google API query.

Usage

```
access_result(
  res,
  result = c("instructions", "routes", "legs", "steps", "points", "polyline",
   "coordinates", "address", "address_components", "geo_place_id", "dist_origins",
    "dist_destinations", "elevation", "elev_location", "place", "place_name",
    "next_page", "place_location", "place_type", "place_hours", "place_open")
)

direction_instructions(res)

direction_routes(res)

direction_legs(res)

direction_steps(res)

direction_points(res)

direction_polyline(res)

distance_origins(res)
```

```
distance_destinations(res)

distance_elements(res)

elevation(res)

elevation_location(res)

geocode_coordinates(res)

geocode_address(res)

geocode_address_components(res)

geocode_place(res)

geocode_type(res)

place(res)

place_next_page(res)

place_name(res)

place_location(res)

place_type(res)

place_hours(res)

place_open(res)

nearest_roads_coordinates(res)
```

Arguments

res	result from a Google API query
result	the specific field of the result you want to access

Functions

- direction_instructions: the instructions from a directions query
- direction_routes: the routes from a directions query
- direction_legs: the legs from a directions query
- direction_steps: the steps from a directions query
- direction_points: the points from a directions query

- direction_polyline: the encoded polyline from a direction query
- distance_origins: the origin addresses from a distance query
- distance_destinations: the destination addresses from a distance query
- distance_elements: the element results from a distance query
- elevation: the elevation from an elevation query
- elevation_location: the elevation from an elevation query
- geocode_coordinates: the coordinates from a geocode or reverse geocode query
- geocode_address: the formatted address from a geocode or reverse geocode query
- geocode_address_components: the address components from a geocode or reverse geocode query
- geocode_place: the place id from a geocode or reverse geocode query
- geocode_type: the geocoded place types from a geocode or reverse geocode query
- place: the place_id from a places query
- place_next_page: the next page token from a places query
- place_name: the place name from a places query
- place_location: the location from a places query
- place_type: the type of place from a places query
- place_hours: the opening hours from a place details query
- place_open: the open now result from a place details query
- nearest_roads_coordinates: the coordinates from a nearest roads query

Examples

```
## Not run:

apiKey <- "your_api_key"

## results returned as a list (simplify == TRUE)
lst <- google_directions(origin = c(-37.8179746, 144.9668636),
                         destination = c(-37.81659, 144.9841),
                         mode = "walking",
                         key = apiKey,
                         simplify = TRUE)

## results returned as raw JSON character vector
js <- google_directions(origin = c(-37.8179746, 144.9668636),
                         destination = c(-37.81659, 144.9841),
                         mode = "walking",
                         key = apiKey,
                         simplify = FALSE)

access_result(js, "polyline")

direction_polyline(js)


## End(Not run)
```

add_bicycling *Add bicycling*

Description

Adds bicycle route information to a googleway map object

Usage

```
add_bicycling(map)
```

Arguments

map	a googleway map object created from google_map()

Examples

```
## Not run:

map_key <- "your_api_key"
google_map(key = map_key) %>%
  add_bicycling()


## End(Not run)
```

add_circles *Add circle*

Description

Add circles to a google map

Usage

```
add_circles(
  map,
  data = get_map_data(map),
  id = NULL,
  lat = NULL,
  lon = NULL,
  polyline = NULL,
  radius = NULL,
  editable = NULL,
  draggable = NULL,
  stroke_colour = NULL,
```

```
  stroke_opacity = NULL,
  stroke_weight = NULL,
  fill_colour = NULL,
  fill_opacity = NULL,
  mouse_over = NULL,
  mouse_over_group = NULL,
  info_window = NULL,
  layer_id = NULL,
  update_map_view = TRUE,
  z_index = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL,
  load_interval = 0,
  focus_layer = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
id	string specifying the column containing an identifier for a shape
lat	string specifying the column of data containing the 'latitude' coordinates. If left NULL, a best-guess will be made
lon	string specifying the column of data containing the 'longitude' coordinates. If left NULL, a best-guess will be made
polyline	string specifying the column of data containing the encoded polyline. For circles and markers the encoded string will represent a single point.
radius	either a string specifying the column of data containing the radius of each circle, OR a numeric value specifying the radius of all the circles (radius is expressed in metres)
editable	string specifying the column of data defining if the polygon is 'editable' (either TRUE or FALSE)
draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes

fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
mouse_over	string specifying the column of data to display when the mouse rolls over the shape
mouse_over_group	string specifying the column of data specifying which groups of shapes to highlight on mouseover
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
update_map_view	logical specifying if the map should re-centre according to the shapes
z_index	single value specifying where the circles appear in the layering of the map objects. Layers with a higher z_index appear on top of those with a lower z_index. See details.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.
load_interval	time in miliseconds to wait between plotting each shape
focus_layer	logical indicating if the map should re-centre according to this layer

Details

z_index values define the order in which objects appear on the map. Those with a higher value appear on top of those with a lower value. The default order of objects is (1 being underneath all other objects)

- 1. Polygon
- 2. Rectangle
- 3. Polyline
- 4. Circle

Markers are always the top layer

palette

The `palette` is used to specify the colours that will map to variables. You can specify a single function to map to all variables, or a named list that specifies a separate function to map to each variable. The elements must be named either `fill_colour` or `stroke_colour`, and their values are the colour generating functions. The default is `viridisLite::viridis`

The `legend_options` can be used to control the appearance of the legend. This should be a named list, where the names are one of

- position - one of `c("TOP_LEFT", "TOP_CENTER", "TOP_RIGHT", "RIGHT_TOP", "RIGHT_CENTER", "RIGHT_BOTTOM", "BOTTOM_RIGHT", "BOTTOM_CENTER", "BOTTOM_LEFT", "LEFT_BOTTOM", "LEFT_CENTER", "LEFT_TOP")`
- css - a string of valid `css` for controlling the appearance of the legend
- title - a string to use for the title of the legend

if `legend_options` are NULL, the default values will apply

If you are displaying two legends, one for `stroke_colour` and one for `fill_colour`, you can specify different options for the different colour attributes. See examples for add_circles.

Examples

```
## Not run:

map_key <- 'your_api_key'

google_map(key = map_key, data = tram_stops) %>%
 add_circles(lat = "stop_lat", lon = "stop_lon", fill_colour = "stop_name",
 stroke_weight = 0.3, stroke_colour = "stop_name", info_window ="stop_id")

## different colour palettes
lstPalette <- list(fill_colour = colorRampPalette(c("red","blue")),
    stroke_colour = viridisLite::plasma)

## set the key via set_key()
set_key(key = map_key)

google_map(data = tram_stops) %>%
 add_circles(lat = "stop_lat", lon = "stop_lon", fill_colour = "stop_lat",
 stroke_weight = 2, stroke_colour = "stop_name", palette = lstPalette, legend = T)

## controlling the legend
google_map(data = tram_stops) %>%
 add_circles(lat = "stop_lat", lon = "stop_lon", fill_colour = "stop_lat",
 stroke_weight = 2, stroke_colour = "stop_name",
 legend = c(fill_colour = T, stroke_colour = F),
 legend_options = list(position = "TOP_RIGHT", css = "max-height: 100px;"))

google_map(data = tram_stops) %>%
 add_circles(lat = "stop_lat", lon = "stop_lon", fill_colour = "stop_lat",
 stroke_weight = 2, stroke_colour = "stop_name",
 legend = T,
```

```
 legend_options = list(
   fill_colour = list(position = "TOP_RIGHT", css = "max-height: 100px;"),
   stroke_colour = list(position = "LEFT_BOTTOM", title = "Stop Name")
   ))



## End(Not run)
```

---

add_dragdrop *Drag Drop Geojson*

Description

A function that enables you to drag data and drop it onto a map. Currently only supports GeoJSON files / text

Usage

```
add_dragdrop(map)
```

Arguments

map	a googleway map object created from google_map()

---

add_drawing *Add Drawing*

Description

Adds drawing tools to the map. Particularly useful when in an interactive (shiny) environment.

Usage

```
add_drawing(
  map,
  drawing_modes = c("marker", "circle", "polygon", "polyline", "rectangle"),
  delete_on_change = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
drawing_modes	string vector giving the drawing controls required. One of one or more of marker, circle, polygon, polyline and rectangle
delete_on_change	logical indicating if the currently drawn shapes should be deleted when a new drawing mode is selected (only works in a reactive environment)

Examples

```
## Not run:

map_key <- 'your_api_key'
google_map(key = map_key) %>%
  add_drawing()


## End(Not run)
```

`add_geojson` *Add GeoJson*

Description

Add GeoJson

Usage

```
add_geojson(
  map,
  data = get_map_data(map),
  layer_id = NULL,
  style = NULL,
  mouse_over = FALSE,
  update_map_view = TRUE
)
```

Arguments

`map`	a googleway map object created from `google_map()`
`data`	A character string or geoJSON literal of correctly formatted geoJSON
`layer_id`	single value specifying an id for the layer.
`style`	Style options for the geoJSON. See details
`mouse_over`	logical indicating if a feature should be highlighted when the mouse passess over
`update_map_view`	logical specifying if the map should re-centre according to the geoJSON

Details

The style of the geoJSON features can be defined inside the geoJSON itself, or specified as a JSON string or R list that's used to style all the features the same

To use the properties in the geoJSON to define the styles, set the `style` argument to a JSON string or a named list, where each name is one of

All Geometries

- clickable
- visible
- zIndex

Point Geometries

- cursor
- icon
- shape
- title

Line Geometries

- strokeColor
- strokeOpacity
- strokeWeight

Polygon Geometries (Line Geometries, plus)

- fillColor
- fillOpacity

and where the values are the the properties of the geoJSON that contain the relevant style for those properties.

To style all the features the same, supply a JSON string or R list that defines a value for each of the style options (listed above)

See examples.

Examples

```
## Not run:


## use the properties inside the geoJSON to style each feature
google_map(key = map_key) %>%
  add_geojson(data = geo_melbourne)

## use a JSON string to style all features
style <- '{ "fillColor" : "green" , "strokeColor" : "black", "strokeWeight" : 0.5}'
google_map(key = map_key) %>%
  add_geojson(data = geo_melbourne, style = style)


## use a named list to style all features
style <- list(fillColor = "red" , strokeColor = "blue", strokeWeight = 0.5)
google_map(key = map_key) %>%
  add_geojson(data = geo_melbourne, style = style)

## GeoJSON from a URL
```

```
url <- 'https://storage.googleapis.com/mapsdevsite/json/google.json'
google_map(key = map_key) %>%
  add_geojson(data = url, mouse_over = T)


## End(Not run)
```

`add_heatmap` *Add heatmap*

Description

Adds a heatmap to a google map

Usage

```
add_heatmap(
  map,
  data = get_map_data(map),
  lat = NULL,
  lon = NULL,
  weight = NULL,
  option_gradient = NULL,
  option_dissipating = FALSE,
  option_radius = 0.01,
  option_opacity = 0.6,
  layer_id = NULL,
  update_map_view = TRUE,
  digits = 4,
  legend = F,
  legend_options = NULL
)
```

Arguments

`map`	a googleway map object created from `google_map()`
`data`	data frame containing the data to use in the layer. If Null, the data passed into `google_map()` will be used.
`lat`	string specifying the column of `data` containing the 'latitude' coordinates. If left NULL, a best-guess will be made
`lon`	string specifying the column of `data` containing the 'longitude' coordinates. If left NULL, a best-guess will be made
`weight`	string specifying the column of `data` containing the 'weight' associated with each point. If NULL, each point will get a weight of 1.
`option_gradient`	vector of colours to use as the gradient colours. see Details

option_dissipating
: logical Specifies whether heatmaps dissipate on zoom. When dissipating is FALSE the radius of influence increases with zoom level to ensure that the color intensity is preserved at any given geographic location. When set to TRUE you will likely need a greater option_radius value. Defaults to FALSE.

option_radius
: numeric. The radius of influence for each data point, in pixels. Defaults to 0.01

option_opacity
: The opacity of the heatmap, expressed as a number between 0 and 1. Defaults to 0.6.

layer_id
: single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.

update_map_view
: logical specifying if the map should re-centre according to the shapes

digits
: integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.

legend
: either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.

legend_options
: A list of options for controlling the legend.

Details

The legend will only show if you supply a weight variable.

option_gradient colours can be two of the R colour specifications; either a colour name (as listed by colors(), or a hexadecimal string of the form "#rrggbb").

The first colour in the vector will be used as the colour that fades to transparent, and is not actually mapped to any data points (and therefore won't be included in the legend). The last colour in the vector will be use in the centre of the 'heat'.

The option_gradient, option_dissipating, option_radius and option_opacity values apply to all points in the data.8

Examples

```
## Not run:

map_key <- 'your_api_key'

set.seed(20170417)
df <- tram_route
df$weight <- sample(1:10, size = nrow(df), replace = T)

google_map(key = map_key, data = df) %>%
 add_heatmap(lat = "shape_pt_lat", lon = "shape_pt_lon", weight = "weight",
              option_radius = 0.001, legend = T)

## specifying different colour gradient
option_gradient <- c('orange', 'blue', 'mediumpurple4', 'snow4', 'thistle1')

google_map(key = map_key, data = df) %>%
 add_heatmap(lat = "shape_pt_lat", lon = "shape_pt_lon", weight = "weight",
```

```
                option_radius = 0.001, option_gradient = option_gradient, legend = T)

## End(Not run)
```

---

add_kml *Add KML*

Description

Adds a KML layer to a map.

Usage

```
add_kml(map, kml_url, layer_id = NULL, update_map_view = TRUE, z_index = 5)
```

Arguments

map	a googleway map object created from google_map()
kml_url	URL string specifying the location of the kml layer
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
update_map_view	logical specifying if the map should re-centre according to the shapes
z_index	single value specifying where the circles appear in the layering of the map objects. Layers with a higher z_index appear on top of those with a lower z_index. See details.

Examples

```
## Not run:

map_key <- 'your_api_key'

kmlUrl <- paste0('https://developers.google.com/maps/',
'documentation/javascript/examples/kml/westcampus.kml')

google_map(key = map_key) %>%
  add_kml(kml_url = kmlUrl)


## End(Not run)
```

add_markers *Add markers*

Description

Add markers to a google map

Usage

```
add_markers(
  map,
  data = get_map_data(map),
  id = NULL,
  colour = NULL,
  lat = NULL,
  lon = NULL,
  polyline = NULL,
  title = NULL,
  draggable = NULL,
  opacity = NULL,
  label = NULL,
  info_window = NULL,
  mouse_over = NULL,
  mouse_over_group = NULL,
  marker_icon = NULL,
  layer_id = NULL,
  cluster = FALSE,
  cluster_options = list(),
  update_map_view = TRUE,
  digits = 4,
  load_interval = 0,
  focus_layer = FALSE,
  close_info_window = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
id	string specifying the column containing an identifier for a shape
colour	string specifying the column containing the 'colour' to use for the markers. One of 'red', 'blue', 'green' or 'lavender'.
lat	string specifying the column of data containing the 'latitude' coordinates. If left NULL, a best-guess will be made

lon	string specifying the column of data containing the 'longitude' coordinates. If left NULL, a best-guess will be made
polyline	string specifying the column of data containing the encoded polyline. For circles and markers the encoded string will represent a single point.
title	string specifying the column of data containing the 'title' of the markers. The title is displayed when you hover over a marker. If blank, no title will be displayed for the markers.
draggable	string specifying the column of data defining if the marker is 'draggable' (either TRUE or FALSE)
opacity	string specifying the column of data defining the 'opacity' of the maker. Values must be between 0 and 1 (inclusive).
label	string specifying the column of data defining the character to appear in the centre of the marker. Values will be coerced to strings, and only the first character will be used.
info_window	string specifying the column of data to display in an info window when a shape is clicked.
mouse_over	string specifying the column of data to display when the mouse rolls over the shape
mouse_over_group	string specifying the column of data specifying which groups of shapes to highlight on mouseover
marker_icon	string specifying the column of data containing a link/URL to an image to use for a marker
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
cluster	logical indicating if co-located markers should be clustered when the map zoomed out
cluster_options	list of options used in clustering. See details.
update_map_view	logical specifying if the map should re-centre according to the shapes
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
load_interval	time in miliseconds to wait between plotting each shape
focus_layer	logical indicating if the map should re-centre according to this layer
close_info_window	logical indicating if all info_windows should close when the user clicks on the map

Details

Cluster Options can be supplied as a named list. The available names are

- gridSize (number) - The grid size of a cluster in pixels

- maxZoom (number) - The maximum zoom level that a marker can be part of a cluster
- zoomOnClick (logical) - Whether the default behaviour of clicking on a cluster is to zoom into it
- averageCenter (logical) - Whether the center of each cluster should be the average of all markers in the cluster
- minimumClusterSize (number) - The minimum number of markers required for a cluster

opts <- list(minimumClusterSize = 3)

Examples

```
## Not run:

map_key <- "your api key"

google_map(
  key = map_key
  , data = tram_stops
  ) %>%
 add_markers(
   lat = "stop_lat"
   , lon = "stop_lon"
   , info_window = "stop_name"
   )


## using marker icons
iconUrl <- paste0("https://developers.google.com/maps/documentation/",
"javascript/examples/full/images/beachflag.png")

tram_stops$icon <- iconUrl

google_map(
  key = map_key
  , data = tram_stops
  ) %>%
  add_markers(
    lat = "stop_lat"
    , lon = "stop_lon"
    , marker_icon = "icon"
  )

## Clustering
google_map(
  key = map_key
  , data = tram_stops
  ) %>%
 add_markers(
   lat = "stop_lat"
   , lon = "stop_lon"
   , info_window = "stop_name"
```

```
    , cluster = TRUE
    , cluster_options = list( minimumClusterSize = 5 )
    )


## End(Not run)
```

`add_overlay` *Add Overlay*

Description

Adds a ground overlay to a map. The overlay can only be added from a URL

Usage

```
add_overlay(
  map,
  north,
  east,
  south,
  west,
  overlay_url,
  layer_id = NULL,
  digits = 4,
  update_map_view = TRUE
)
```

Arguments

`map`	a googleway map object created from `google_map()`
`north`	northern-most latitude coordinate
`east`	eastern-most longitude
`south`	southern-most latitude coordinate
`west`	western-most longitude
`overlay_url`	URL string specifying the location of the overlay layer
`layer_id`	single value specifying an id for the layer.
`digits`	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
`update_map_view`	logical. Use this parameter to specify if the map should re-centre according to the overlay extent.

Examples

```
## Not run:

map_key <- 'your_api_key'

google_map(key = map_key) %>%
  add_overlay(north = 40.773941, south = 40.712216, east = -74.12544, west = -74.22655,
           overlay_url = "https://www.lib.utexas.edu/maps/historical/newark_nj_1922.jpg")


url <- paste0("https://developers.google.com/maps/documentation/javascript",
  "/examples/full/images/talkeetna.png")

google_map(key = map_key) %>%
  add_overlay(north = 62.400471, south = 62.281819, east = -150.005608, west = -150.287132,
                overlay_url = url)



## End(Not run)
```

---

add_polygons *Add polygon*

Description

Add a polygon to a google map.

Usage

```
add_polygons(
  map,
  data = get_map_data(map),
  polyline = NULL,
  lat = NULL,
  lon = NULL,
  id = NULL,
  pathId = NULL,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  fill_colour = NULL,
  fill_opacity = NULL,
  info_window = NULL,
  mouse_over = NULL,
  mouse_over_group = NULL,
  draggable = NULL,
  editable = NULL,
```

```
  update_map_view = TRUE,
  layer_id = NULL,
  z_index = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL,
  load_interval = 0,
  focus_layer = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing at least a polyline column, or a lat and a lon column. If Null, the data passed into google_map() will be used.
polyline	string specifying the column of data containing the encoded polyline
lat	string specifying the column of data containing the 'latitude' coordinates. If left NULL, a best-guess will be made
lon	string specifying the column of data containing the 'longitude' coordinates. If left NULL, a best-guess will be made
id	string specifying the column containing an identifier for a shape
pathId	string specifying the column containing an identifer for each path that forms the complete polygon. Not required when using polyline, as each polyline is itself a path.
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.
mouse_over	string specifying the column of data to display when the mouse rolls over the shape
mouse_over_group	string specifying the column of data specifying which groups of shapes to highlight on mouseover

draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
editable	string specifying the column of data defining if the polygon is 'editable' (either TRUE or FALSE)
update_map_view	logical specifying if the map should re-centre according to the shapes
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
z_index	single value specifying where the circles appear in the layering of the map objects. Layers with a higher z_index appear on top of those with a lower z_index. See details.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.
load_interval	time in miliseconds to wait between plotting each shape
focus_layer	logical indicating if the map should re-centre according to this layer

Details

z_index values define the order in which objects appear on the map. Those with a higher value appear on top of those with a lower value. The default order of objects is (1 being underneath all other objects)

- 1. Polygon
- 2. Rectangle
- 3. Polyline
- 4. Circle

Markers are always the top layer

palette

The palette is used to specify the colours that will map to variables. You can specify a single function to map to all variables, or a named list that specifies a separate function to map to each variable. The elements must be named either fill_colour or stroke_colour, and their values are the colour generating functions. The default is viridisLite::viridis

The legend_options can be used to control the appearance of the legend. This should be a named list, where the names are one of

- position - one of c("TOP_LEFT", "TOP_CENTER", "TOP_RIGHT", "RIGHT_TOP", "RIGHT_CENTER", "RIGHT_BOTTOM", "BOTTOM_RIGHT", "BOTTOM_CENTER", "BOTTOM_LEFT", "LEFT_BOTTOM", "LEFT_CENTER", "LEFT_TOP")

- css - a string of valid `css` for controlling the appearance of the legend
- title - a string to use for the title of the legend

if `legend_options` are NULL, the default values will apply

If you are displaying two legends, one for `stroke_colour` and one for `fill_colour`, you can specify different options for the different colour attributes. See examples for add_circles.

Note

A polygon represents an area enclosed by a closed path. Polygon objects are similar to polylines in that they consist of a series of coordinates in an ordered sequence. Polygon objects can describe complex shapes, including

- Multiple non-contiguous areas defined by a single polygon
- Areas with holes in them
- Intersections of one or more areas

To define a complex shape, you use a polygon with multiple paths.

To create a hole in a polygon, you need to create two paths, one inside the other. To create the hole, the coordinates of the inner path must be wound in the opposite order to those defining the outer path. For example, if the coordinates of the outer path are in clockwise order, then the inner path must be anti-clockwise.

You can represent a polygon in one of three ways

- as a series of coordinates defining a path (or paths) with both an `id` and `pathId` argument that make up the polygon
- as an encoded polyline using an `id` column to specify multiple polylines for a polygon
- as a list column in a data.frame, where each row of the data.frame contains the polylines that comprise the polygon

See Examples

See Also

encode_pl

Examples

```
## Not run:

map_key <- 'your_api_key'

## polygon with a hole - Bermuda triangle
## using one row per polygon, and a list-column of encoded polylines
pl_outer <- encode_pl(lat = c(25.774, 18.466,32.321),
      lon = c(-80.190, -66.118, -64.757))

pl_inner <- encode_pl(lat = c(28.745, 29.570, 27.339),
       lon = c(-70.579, -67.514, -66.668))
```

```
df <- data.frame(id = c(1, 1),
       polyline = c(pl_outer, pl_inner),
       stringsAsFactors = FALSE)

df <- aggregate(polyline ~ id, data = df, list)

google_map(key = map_key, height = 800) %>%
    add_polygons(data = df, polyline = "polyline")

## the same polygon, but using an 'id' to specify the polygon
df <- data.frame(id = c(1,1),
       polyline = c(pl_outer, pl_inner),
       stringsAsFactors = FALSE)

google_map(key = map_key, height = 800) %>%
    add_polygons(data = df, polyline = "polyline", id = "id")

## the same polygon, specified using coordinates, and with a second independent
## polygon
df <- data.frame(myId = c(1,1,1,1,1,1,2,2,2),
      lineId = c(1,1,1,2,2,2,1,1,1),
      lat = c(26.774, 18.466, 32.321, 28.745, 29.570, 27.339, 22, 23, 22),
      lon = c(-80.190, -66.118, -64.757, -70.579, -67.514, -66.668, -50, -49, -51),
      colour = c(rep("#00FF0F", 6), rep("#FF00FF", 3)),
      stringsAsFactors = FALSE)

google_map(key = map_key) %>%
  add_polygons(data = df, lat = 'lat', lon = 'lon', id = 'myId', pathId = 'lineId',
               fill_colour = 'colour')




## End(Not run)
```

---

add_polylines *Add polyline*

Description

Add a polyline to a google map

Usage

```
add_polylines(
  map,
  data = get_map_data(map),
  polyline = NULL,
```

```
  lat = NULL,
  lon = NULL,
  id = NULL,
  geodesic = NULL,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  info_window = NULL,
  mouse_over = NULL,
  mouse_over_group = NULL,
  draggable = NULL,
  editable = NULL,
  update_map_view = TRUE,
  layer_id = NULL,
  z_index = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL,
  load_interval = 0,
  focus_layer = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing at least a polyline column, or a lat and a lon column. If Null, the data passed into google_map() will be used.
polyline	string specifying the column of data containing the encoded polyline
lat	string specifying the column of data containing the 'latitude' coordinates. If left NULL, a best-guess will be made
lon	string specifying the column of data containing the 'longitude' coordinates. If left NULL, a best-guess will be made
id	string specifying the column containing an identifier for a shape
geodesic	logical
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.

mouse_over	string specifying the column of data to display when the mouse rolls over the shape
mouse_over_group	string specifying the column of data specifying which groups of shapes to highlight on mouseover
draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
editable	string specifying the column of data defining if the polygon is 'editable' (either TRUE or FALSE)
update_map_view	logical specifying if the map should re-centre according to the shapes
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
z_index	single value specifying where the circles appear in the layering of the map objects. Layers with a higher z_index appear on top of those with a lower z_index. See details.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.
load_interval	time in miliseconds to wait between plotting each shape
focus_layer	logical indicating if the map should re-centre according to this layer

Details

z_index values define the order in which objects appear on the map. Those with a higher value appear on top of those with a lower value. The default order of objects is (1 being underneath all other objects)

- 1. Polygon
- 2. Rectangle
- 3. Polyline
- 4. Circle

Markers are always the top layer

palette

The palette is used to specify the colours that will map to variables. You can specify a single function to map to all variables, or a named list that specifies a separate function to map to each variable. The elements must be named either fill_colour or stroke_colour, and their values are the colour generating functions. The default is viridisLite::viridis

The legend_options can be used to control the appearance of the legend. This should be a named list, where the names are one of

- position - one of `c("TOP_LEFT", "TOP_CENTER", "TOP_RIGHT", "RIGHT_TOP", "RIGHT_CENTER", "RIGHT_BOTTOM", "BOTTOM_RIGHT", "BOTTOM_CENTER", "BOTTOM_LEFT", "LEFT_BOTTOM", "LEFT_CENTER", "LEFT_TOP")`
- css - a string of valid `css` for controlling the appearance of the legend
- title - a string to use for the title of the legend

if `legend_options` are NULL, the default values will apply

If you are displaying two legends, one for `stroke_colour` and one for `fill_colour`, you can specify different options for the different colour attributes. See examples for add_circles.

Note

The lines can be generated by either using an encoded polyline, or by a set of lat/lon coordinates. You sould specify either the column containing an encoded polyline, OR the lat / lon colulmns.

Using `update_map_view = TRUE` for multiple polylines may be slow, so it may be more appropriate to set the view of the map using the location argument of `google_map()`

Examples

```
## Not run:

## using lat/lon coordinates

set_key("your_api_key")

google_map(data = tram_route) %>%
  add_polylines(lat = "shape_pt_lat", lon = "shape_pt_lon")


google_map() %>%
  add_polylines(data = melbourne, polyline = "polyline", stroke_weight = 1,
      stroke_colour = "SA4_NAME")

## using encoded polyline and various colour / fill options
url <- 'https://raw.githubusercontent.com/plotly/datasets/master/2011_february_aa_flight_paths.csv'
flights <- read.csv(url)
flights$id <- seq_len(nrow(flights))


## encode the routes as polylines
lst <- lapply(unique(flights$id), function(x){
 lat = c(flights[flights["id"] == x, c("start_lat")], flights[flights["id"] == x, c("end_lat")])
 lon = c(flights[flights["id"] == x, c("start_lon")], flights[flights["id"] == x, c("end_lon")])
  data.frame(id = x, polyline = encode_pl(lat = lat, lon = lon))
})

flights <- merge(flights, do.call(rbind, lst), by = "id")

style <- map_styles()$night

google_map(key = map_key, style = style) %>%
```

```
  add_polylines(data = flights, polyline = "polyline", mouse_over_group = "airport1",
                stroke_weight = 1, stroke_opacity = 0.3, stroke_colour = "#ccffff")


## End(Not run)
```

---

add_rectangles *Add Rectangles*

Description

Adds a rectangle to a google map

Usage

```
add_rectangles(
  map,
  data = get_map_data(map),
  north,
  east,
  south,
  west,
  id = NULL,
  draggable = NULL,
  editable = NULL,
  stroke_colour = NULL,
  stroke_opacity = NULL,
  stroke_weight = NULL,
  fill_colour = NULL,
  fill_opacity = NULL,
  mouse_over = NULL,
  mouse_over_group = NULL,
  info_window = NULL,
  layer_id = NULL,
  update_map_view = TRUE,
  z_index = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL,
  load_interval = 0,
  focus_layer = FALSE
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the bounds for the rectangles
north	String specifying the column of data that contains the northern most latitude coordinate
east	String specifying the column of data that contains the eastern most longitude
south	String specifying the column of data that contains the southern most latitude coordinate
west	String specifying the column of data that contains the western most longitude
id	string specifying the column containing an identifier for a shape
draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
editable	String specifying the column of data that indicates if the rectangle is editable. The value in the column should be logical
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
mouse_over	string specifying the column of data to display when the mouse rolls over the shape
mouse_over_group	string specifying the column of data specifying which groups of shapes to highlight on mouseover
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
update_map_view	logical specifying if the map should re-centre according to the shapes
z_index	single value specifying where the circles appear in the layering of the map objects. Layers with a higher z_index appear on top of those with a lower z_index. See details.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.

palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.
load_interval	time in miliseconds to wait between plotting each shape
focus_layer	logical indicating if the map should re-centre according to this layer

Details

z_index values define the order in which objects appear on the map. Those with a higher value appear on top of those with a lower value. The default order of objects is (1 being underneath all other objects)

- 1. Polygon
- 2. Rectangle
- 3. Polyline
- 4. Circle

Markers are always the top layer

palette

The palette is used to specify the colours that will map to variables. You can specify a single function to map to all variables, or a named list that specifies a separate function to map to each variable. The elements must be named either fill_colour or stroke_colour, and their values are the colour generating functions. The default is viridisLite::viridis

The legend_options can be used to control the appearance of the legend. This should be a named list, where the names are one of

- position - one of c("TOP_LEFT", "TOP_CENTER", "TOP_RIGHT", "RIGHT_TOP", "RIGHT_CENTER", "RIGHT_BOTTOM", "BOTTOM_RIGHT", "BOTTOM_CENTER", "BOTTOM_LEFT", "LEFT_BOTTOM", "LEFT_CENTER", "LEFT_TOP")
- css - a string of valid css for controlling the appearance of the legend
- title - a string to use for the title of the legend

if legend_options are NULL, the default values will apply

If you are displaying two legends, one for stroke_colour and one for fill_colour, you can specify different options for the different colour attributes. See examples for add_circles.

Examples

```
## Not run:

map_key <- 'your_api_key'

df <- data.frame(north = 33.685, south = 33.671, east = -116.234, west = -116.251)
```

```
google_map(key = map_key) %>%
  add_rectangles(data = df, north = 'north', south = 'south',
                 east = 'east', west = 'west')

## editable rectangle
df <- data.frame(north = -37.8459, south = -37.8508, east = 144.9378,
                  west = 144.9236, editable = T, draggable = T)

google_map(key = map_key) %>%
  add_rectangles(data = df, north = 'north', south = 'south',
                 east = 'east', west = 'west',
                 editable = 'editable', draggable = 'draggable')


## End(Not run)
```

`add_traffic` *Add Traffic*

Description

Adds live traffic information to a googleway map object

Usage

```
add_traffic(map)
```

Arguments

`map`	a googleway map object created from `google_map()`

Examples

```
## Not run:

map_key <- 'your_api_key'
google_map(key = map_key) %>%
  add_traffic()


## End(Not run)
```

add_transit *Add transit*

Description

Adds public transport information to a googleway map object

Usage

```
add_transit(map)
```

Arguments

map	a googleway map object created from google_map()

Examples

```
## Not run:

map_key <- 'your_api_key'
google_map(key = map_key) %>%
  add_transit()


## End(Not run)
```

clear_bounds *Clear bounds*

Description

A helper function to clear the javascript array of lat/lon bounds.

Usage

```
clear_bounds(map)
```

Arguments

map	a googleway map object created from google_map()

clear_circles *Remove drawing*

Description

clears elements from a map

Usage

```
clear_circles(map, layer_id = NULL)

clear_drawing(map)

remove_drawing(map)

clear_fusion(map, layer_id = NULL)

clear_geojson(map, layer_id = NULL)

clear_heatmap(map, layer_id = NULL)

clear_kml(map, layer_id = NULL)

clear_markers(map, layer_id = NULL)

clear_overlay(map, layer_id = NULL)

clear_polygons(map, layer_id = NULL)

clear_polylines(map, layer_id = NULL)

clear_rectangles(map, layer_id = NULL)

clear_traffic(map)

clear_transit(map)

clear_bicycling(map)
```

Arguments

map	a googleway map object created from google_map()
layer_id	id value of the layer to be removed from the map

Functions

- remove_drawing: removes drawing controls from a map

Note

These operations are intended for use in conjunction with google_map_update in an interactive shiny environment

---

`clear_keys` *Clear Keys*

Description

Clears all the API keys

Usage

```
clear_keys()
```

`clear_search` *Clear search*

Description

clears the markers placed on the map after using the search box

Usage

```
clear_search(map)
```

Arguments

`map`	a googleway map object created from `google_map()`

`decode_pl` *Decode PL*

Description

Decodes an encoded polyline into the series of lat/lon coordinates that specify the path

Usage

```
decode_pl(encoded)
```

Arguments

`encoded`	String. An encoded polyline

Value

data.frame of lat/lon coordinates

Note

An encoded polyline is generated from google's polyline encoding algorithm (https://developers.google.com/maps/documentation/utilities/polylinealgorithm).

See Also

encode_pl, google_directions

Examples

```
## polyline joining the capital cities of Australian states
pl <- "nnseFmpzsZgalNytrXetrG}krKsaif@kivIccvzAvvqfClp~uBlymzA~ocQ}_}iCthxo@srst@"

df_polyline <- decode_pl(pl)
df_polyline
```

---

encode_pl *Encode PL*

Description

Encodes a series of lat/lon coordinates that specify a path into an encoded polyline

Usage

```
encode_pl(lat, lon)
```

Arguments

lat	vector of latitude coordinates
lon	vector of longitude coordinates

Value

string encoded polyline

Note

An encoded polyline is generated from google's polyline encoding algorithm (https://developers.google.com/maps/documentation/utilities/polylinealgorithm).

See Also

decode_pl

Examples

```
encode_pl(lat = c(38.5, 40.7, 43.252), lon = c(-120.2, -120.95, -126.453))
## "_p~iF~ps|U_ulLnnqC_mqNvxq`@"
```

`geo_melbourne` *geo_melbourne*

Description

GeoJSON data of Melbourne's Inner suburbs.

Usage

```
geo_melbourne
```

Format

An object of class `json` (inherits from `geo_json`) of length 1.

Details

This is a subset of the `melbourne` data.

`google_charts` *Google Charts*

Description

Google Charts can be displayed inside an `info_window`

`info_window`

When using a chart in an `info_window` you need to use a `list` with at least two elements named `data` and `type`. You can also use a third element called `options` for controlling the appearance of the chart.

You must also supply the `id` argument to the layer your are adding (e.g. `add_markers()`), and the `data` must have a column with the same name as the `id` (and therefore the same name as the `id` column in the original `data` supplied to the `add_` function).

See the specific chart sections for details on how to structure the `data`.

chart types

the `type` element can be one of

- `area`
- `bar`
- `bubble`
- `candlestick`
- `column`
- `combo`
- `histogram`
- `line`
- `pie`
- `scatter`

Area

data

An area chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `area`

options see the area charts documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/areachart

Each row of data represents a data point at the same x-axis location

Bar

data

A bar chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `bar`

options

See the bar chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/barchart

Bubble

data

A bubble chart requires a `data.frame` of at least four, and at most six columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third column: x-axis value
4. Fourth column: y-axis value
5. Fith column: visualised as colour
6. Sixth column: visualised as size

type - `bubble`

options

See the bubble chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/bubblechart

Candlestick

data

A candlestick chart requires a `data.frame` of at least six columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third column: Number specifying the 'low' number for the data
4. Fourth column: Number specifying the opening/initial value of the data. This is one vertical border of the candle. If less than the column 4 value, the candle will be filled; otherwise it will be hollow.
5. Fith column: Number specifying the closing/final value of the data. This is the second vertical border of the candle. If less than the column 3 value, the candle will be hollow; otherwise it will be filled.
6. Sixth column: Number specifying the high/maximum value of this marker. This is the top of the candle's center line.

type - `candlestick`

options

See the candlestick chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/candlestickchart

Column

data

A column chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `column`

options

See the column chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/columnchart

Combo

A combo chart lets you render each series as a different marker type from the following list: line, area, bars, candlesticks, and stepped area.

data

A combo chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `combo`

options

See the column chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/combochart

Histogram

data

A histogram chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `histogram`

options

See the histogram chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/histogram

Line

data

A line chart requires a `data.frame` of at least three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third or more columns: the data used in the chart

type - `line`

options

See the line chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/linechart

Pie

data

A pie chart requires a `data.frame` of three columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third column: the data used in the chart

type - `pie`

options

See the pie chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/piechart

Scatter

data

A scatter chart requires a `data.frame` of at least four columns:

1. First column: a column of id values, where the column has the same name as the id column in the `data` argument, and therefore the same name as the value supplied to the `id` argument.
2. Second column: variable names used for labelling the data
3. Third column: the data plotted on x-axis
4. Fourth or more columns: the data plotted on y-axis

type - `scatter`

options

See the scatter chart documentation for various other examples https://developers.google.com/chart/interactive/docs/gallery/scatterchart

Examples

```
## Not run:

set_key("your_api_key")

## AREA
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 2),
    year = rep( c("year1", "year2")),
    arrivals = sample(1:100, size = nrow(tram_stops) * 2, replace = T),
    departures = sample(1:100, size = nrow(tram_stops) * 2, replace = T))

chartList <- list(data = markerCharts,
   type = 'area',
   options = list(width = 400, chartArea = list(width = "50%")))

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")

tram_route$id <- c(rep(1, 30), rep(2, 25))

lineCharts <- data.frame(id = rep(c(1,2), each = 2),
    year = rep( c("year1", "year2") ),
    arrivals = sample(1:100, size = 4),
    departures = sample(1:100, size = 4))

chartList <- list(data = lineCharts,
   type = 'area')

google_map() %>%
  add_polylines(data = tram_route, id = 'id',
    stroke_colour = "id", stroke_weight = 10,
    lat = "shape_pt_lat", lon = "shape_pt_lon",
    info_window = chartList
    )


## End(Not run)

## Not run:

## BAR
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 2),
    year = rep( c("year1", "year2")),
    arrivals = sample(1:100, size = nrow(tram_stops) * 2, replace = T),
    departures = sample(1:100, size = nrow(tram_stops) * 2, replace = T))

chartList <- list(data = markerCharts,
   type = 'bar')

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")
```

```
lineChart <- data.frame(id = 33,
    year = c("year1","year2"),
    val1 = c(1,2),
    val2 = c(2,1))

chartList <- list(data = lineChart, type = 'bar')

google_map() %>%
  add_polylines(data = melbourne[melbourne$polygonId == 33, ],
  polyline = "polyline",
  info_window = chartList)


## End(Not run)


## Not run:

## BUBBLE
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 4),
    ID = sample(letters, size = nrow(tram_stops) * 4, replace = T),
    time = sample(1:1440, size = nrow(tram_stops) * 4, replace = T),
    passengers = sample(1:100, size = nrow(tram_stops) * 4, replace = T),
    year = c("year1", "year2", "year3", "year4"),
    group = sample(50:100, size = nrow(tram_stops) * 4, replace = T))

chartList <- list(data = markerCharts,
   type = 'bubble')

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)

## Not run:

## CANDLESTICK
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 5),
    day = rep(c("Mon", "Tues", "Weds", "Thurs", "Fri"), times = nrow(tram_stops) ),
    val1 = rep(c(20, 31, 50, 77, 68), times = nrow(tram_stops) ),
    val2 = rep(c(28, 38, 55, 77, 66), times = nrow(tram_stops) ),
    val3 = rep(c(38, 55, 77, 66, 22), times = nrow(tram_stops) ),
    val4 = rep(c(45, 66, 80, 50, 15), times = nrow(tram_stops) ) )

chartList <- list(data = markerCharts,
   type = 'candlestick',
   options = list(legend = 'none',
     bar = list(groupWidth = "100%"),
     candlestick = list(
       fallingColor = list( strokeWidth = 0, fill = "#a52714"),
       risingColor = list( strokeWidth = 0, fill = "#0f9d58")
```

```
       )
     ))

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)


## Not run:

## COLUMN
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 2),
    year = rep( c("year1", "year2")),
    arrivals = sample(1:100, size = nrow(tram_stops) * 2, replace = T),
    departures = sample(1:100, size = nrow(tram_stops) * 2, replace = T))

chartList <- list(data = markerCharts,
   type = 'column')

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")

polyChart <- data.frame(id = 33,
    year = c("year1","year2"),
    val1 = c(1,2),
    val2 = c(2,1))

chartList <- list(data = polyChart, type = 'column')

google_map() %>%
  add_polygons(data = melbourne[melbourne$polygonId == 33, ],
  polyline = "polyline",
  info_window = chartList)

tram_route$id <- 1

polyChart <- data.frame(id = 1,
    year = c("year1","year2"),
    val1 = c(1,2),
    val2 = c(2,1))

chartList <- list(data = polyChart, type = 'column')

google_map() %>%
  add_polygons(data = tram_route,
    lon = "shape_pt_lon", lat = "shape_pt_lat",
    info_window = chartList)



## End(Not run)
```

```
## Not run:


## COMBO
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 2),
    year = rep( c("year1", "year2")),
    arrivals = sample(1:100, size = nrow(tram_stops) * 2, replace = T),
    departures = sample(1:100, size = nrow(tram_stops) * 2, replace = T))

markerCharts$val <- sample(1:100, size = nrow(markerCharts), replace = T)

chartList <- list(data = markerCharts,
   type = 'combo',
   options = list(
     "title" = "Passengers at stops",
     "vAxis" = list( title = "passengers" ),
     "hAxis" = list( title = "load" ),
     "seriesType" = "bars",
     "series" = list( "2" = list( "type" = "line" )))) ## 0-indexed

google_map() %>%
  add_circles(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)


## Not run:

## HISTOGRAM
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 20),
    day = as.character(1:20))

markerCharts$wait <- rnorm(nrow(markerCharts), 0, 1)

chartList <- list(data = markerCharts,
   type = 'histogram')

google_map() %>%
  add_circles(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)


## Not run:

## Line
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 20),
    day = as.character(1:20),
    value = sample(1:100, size = nrow(tram_stops) * 20, replace = T))
```

```
chartList <- list(data = markerCharts,
   type = 'line')

google_map() %>%
  add_circles(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)


## Not run:

## PIE
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 3))
markerCharts$variable <- c("yes", "no", "maybe")
markerCharts$value <- sample(1:10, size = nrow(markerCharts), replace = T)

chartList <- list(data = markerCharts,
   type = 'pie',
   options = list(title = "my pie",
     is3D = TRUE,
     height = 240,
     width = 240,
     colors = c('#440154', '#21908C', '#FDE725')))

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")

## use pieHole option to make a donut chart

chartList <- list(data = markerCharts,
   type = 'pie',
   options = list(title = "my pie",
     pieHole = 0.4,
     height = 240,
     width = 240,
     colors = c('#440154', '#21908C', '#FDE725')))

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")


## End(Not run)


## Not run:

## SCATTER
markerCharts <- data.frame(stop_id = rep(tram_stops$stop_id, each = 5))
markerCharts$arrival <- sample(1:10, size = nrow(markerCharts), replace = T)
markerCharts$departure <- sample(1:10, size = nrow(markerCharts), replace = T)

chartList <- list(data = markerCharts,
```

```
   type = 'scatter')

google_map() %>%
  add_markers(data = tram_stops, info_window = chartList, id = "stop_id")

## End(Not run)
```

`google_directions` *Google Directions*

Description

The Google Maps Directions API is a service that calculates directions between locations. You can search for directions for several modes of transportation, including transit, driving, walking, or cycling.

Usage

```
google_directions(
  origin,
  destination,
  mode = c("driving", "walking", "bicycling", "transit"),
  departure_time = NULL,
  arrival_time = NULL,
  waypoints = NULL,
  optimise_waypoints = FALSE,
  alternatives = FALSE,
  avoid = NULL,
  units = c("metric", "imperial"),
  traffic_model = NULL,
  transit_mode = NULL,
  transit_routing_preference = NULL,
  language = NULL,
  region = NULL,
  key = get_api_key("directions"),
  simplify = TRUE,
  curl_proxy = NULL
)
```

Arguments

`origin`	Origin location as either a one or two column data.frame, a list of unnamed elements, each element is either a numeric vector of lat/lon coordinates, an address string or a place_id, or a vector of a pair of lat / lon coordinates

`destination` destination location as either a one or two column data.frame, a list of unnamed elements, each element is either a numeric vector of lat/lon coordinates, an address string or place_id, or a vector of a pair of lat / lon coordinates

`mode` `string` One of 'driving', 'walking', 'bicycling' or 'transit'.

`departure_time` The desired time of departure. Use either a `POSIXct` time since 1st January 1970, or the string 'now'. If no value is specified it defaults to `Sys.time()`.

`arrival_time` Specifies the desired time of arrival for transit requests. Use either a `POSIXct` time since 1st January 1970. Note you can only specify one of `arrival_time` or `departure_time`, not both. If both are supplied, `departure_time` will be used.

`waypoints` list of waypoints, expressed as either `vectors` of lat/lon coordinates, or a `string` address to be geocoded, or an encoded polyline enclosed by `enc:` and `:`. Only available for driving, walking or bicycling modes. List elements must be named either 'stop' or 'via', where 'stop' is used to indicate a stopover for a waypoint, and 'via' will not stop at the waypoint. See https://developers.google.com/maps/documentation/directions/overview#Waypoints for details

`optimise_waypoints` `boolean` allow the Directions service to optimize the provided route by rearranging the waypoints in a more efficient order. (This optimization is an application of the Travelling Salesman Problem.) Travel time is the primary factor which is optimized, but other factors such as distance, number of turns and many more may be taken into account when deciding which route is the most efficient. All waypoints must be stopovers for the Directions service to optimize their route.

`alternatives` `logical` If set to true, specifies that the Directions service may provide more than one route alternative in the response

`avoid` `character` vector stating which features should be avoided. One of 'tolls', 'highways', 'ferries' or 'indoor'

`units` `string` metric or imperial. Note: Only affects the text displayed within the distance field. The values are always in metric

`traffic_model` `string` - one of 'best_guess', 'pessimistic' or 'optimistic'. Only valid with a departure time

`transit_mode` `vector` of strings, either 'bus', 'subway', 'train', 'tram' or 'rail'. Only vaid where `mode = 'transit'`. Note that 'rail' is equivalent to `transit_mode=c("train", "tram", "subway")`

`transit_routing_preference` `vector` of strings - one of 'less_walking' and 'fewer_transfers'. specifies preferences for transit routes. Only valid for transit directions.

`language` `string` - specifies the language in which to return the results. See the list of supported languages: https://developers.google.com/maps/faq#languagesupport. If no langauge is supplied, the service will attempt to use the language of the domain from which the request was sent

`region` `string` - specifies the region code, specified as a ccTLD ("top-level domain"). See region basing for details https://developers.google.com/maps/documentation/directions/overview#RegionBiasing

`key` `string` - a valid Google Developers Directions API key

simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object

Value

Either list or JSON string of the route between origin and destination

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:

set_key("YOUR_GOOGLE_API_KEY")

## using lat/long coordinates
google_directions(origin = c(-37.8179746, 144.9668636),
          destination = c(-37.81659, 144.9841),
          mode = "walking")


## using address string
google_directions(origin = "Flinders Street Station, Melbourne",
         destination = "MCG, Melbourne",
         mode = "walking")


google_directions(origin = "Melbourne Airport, Australia",
         destination = "Portsea, Melbourne, Australia",
         departure_time =  Sys.time() + (24 * 60 * 60),
         waypoints = list(stop = c(-37.81659, 144.9841),
                          via = "Ringwood, Victoria"),
         mode = "driving",
         alternatives = FALSE,
         avoid = c("TOLLS", "highways"),
         units = "imperial",
         simplify = TRUE)

## using 'now' as departure time
google_directions(origin = "Flinders Street Station, Melbourne",
         destination = "MCG, Melbourne",
         departure_time = 'now')
```

```
## waypoints expressed as an encoded polyline
polyWaypoints <- encode_pl(tram_stops[1:2, c("stop_lat")], tram_stops[1:2, c("stop_lon")])
polyWaypoints <- list(via = paste0("enc:", polyWaypoints, ":"))

google_directions(origin = "Melbourne Zoo, Melbourne",
         destination = "Studley Park, Melbourne",
         waypoints = polyWaypoints)


## using bus and less walking
res <- google_directions(origin = "Melbourne Airport, Australia",
         destination = "Portsea, Melbourne, Australia",
         departure_time =  Sys.time() + (24 * 60 * 60),
         mode = "transit",
         transit_mode = "bus",
         transit_routing_preference = "less_walking",
         simplify = FALSE)

## using arrival time
res <- google_directions(origin = "Melbourne Airport, Australia",
         destination = "Portsea, Melbourne, Australia",
         arrival_time =  Sys.time() + (24 * 60 * 60),
         mode = "transit",
         transit_mode = "bus",
         transit_routing_preference = "less_walking",
         simplify = FALSE)

## return results in French
res <- google_directions(origin = "Melbourne Airport, Australia",
         destination = "Portsea, Melbourne, Australia",
         arrival_time =  Sys.time() + (24 * 60 * 60),
         mode = "transit",
         transit_mode = "bus",
         transit_routing_preference = "less_walking",
         language = "fr",
         simplify = FALSE)


## End(Not run)
```

`google_dispatch` *Google dispatch*

Description

Extension points for plugins

Usage

```
google_dispatch(
```

```
  map,
  funcName,
  google_map = stop(paste(funcName, "requires a map update object")),
 google_map_update = stop(paste(funcName, "does not support map udpate objects"))
)

invoke_method(map, method, ...)
```

Arguments

map	a map object, as returned from google_map
funcName	the name of the function that the user called that caused this google_dispatch call; for error message purposes
google_map	an action to be performed if the map is from google_map
google_map_update	an action to be performed if the map is from google_map_update
method	the name of the JavaScript method to invoke
...	unnamed arguments to be passed to the JavaScript method

Value

google_dispatch returns the value of google_map or or an error. invokeMethod returns the map object that was passed in, possibly modified.

---

google_distance *Google Distance*

Description

The Google Maps Distance Matrix API is a service that provides travel distance and time for a matrix of origins and destinations, based on the recommended route between start and end points.

Usage

```
google_distance(
  origins,
  destinations,
  mode = c("driving", "walking", "bicycling", "transit"),
  departure_time = NULL,
  arrival_time = NULL,
  avoid = NULL,
  units = c("metric", "imperial"),
  traffic_model = NULL,
  transit_mode = NULL,
  transit_routing_preference = NULL,
  language = NULL,
```

```
  key = get_api_key("distance"),
  simplify = TRUE,
  curl_proxy = NULL
)
```

Arguments

origins	Origin locations as either a one or two column data.frame, a list of unnamed elements, each element is either a numeric vector of lat/lon coordinates, an address string or a place_id, or a vector of a pair of lat / lon coordinates
destinations	destination locations as either a one or two column data.frame, a list of unnamed elements, each element is either a numeric vector of lat/lon coordinates, an address string or place_id, or a vector of a pair of lat / lon coordinates
mode	string One of 'driving', 'walking', 'bicycling' or 'transit'.
departure_time	The desired time of departure. Use either a POSIXct time since 1st January 1970, or the string 'now'. If no value is specified it defaults to Sys.time().
arrival_time	Specifies the desired time of arrival for transit requests. Use either a POSIXct time since 1st January 1970. Note you can only specify one of arrival_time or departure_time, not both. If both are supplied, departure_time will be used.
avoid	character vector stating which features should be avoided. One of 'tolls', 'highways', 'ferries' or 'indoor'
units	string metric or imperial. Note: Only affects the text displayed within the distance field. The values are always in metric
traffic_model	string - one of 'best_guess', 'pessimistic' or 'optimistic'. Only valid with a departure time
transit_mode	vector of strings, either 'bus', 'subway', 'train', 'tram' or 'rail'. Only vaid where mode = 'transit'. Note that 'rail' is equivalent to transit_mode=c("train", "tram", "subway")
transit_routing_preference	vector strings - one of 'less_walking' and 'fewer_transfers'. specifies preferences for transit routes. Only valid for transit directions.
language	string - specifies the language in which to return the results. See the list of supported languages: https://developers.google.com/maps/faq#languagesupport. If no langauge is supplied, the service will attempt to use the language of the domain from which the request was sent
key	string - a valid Google Developers Directions API key
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object

Value

Either list or JSON string of the distance between origins and destinations

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:

set_key("YOUR_GOOGLE_API_KEY")
google_distance(origins = list(c("Melbourne Airport, Australia"),
                               c("MCG, Melbourne, Australia"),
                               c(-37.81659, 144.9841)),
                               destinations = c("Portsea, Melbourne, Australia"),
                               simplify = FALSE)

google_distance(origins = c(-37.816, 144.9841),
   destinations = c("Melbourne Airport, Australia", "Flinders Street Station, Melbourne"))

google_distance(origins = tram_stops[1:5, c("stop_lat", "stop_lon")],
     destinations = tram_stops[10:12, c("stop_lat", "stop_lon")],)


## End(Not run)
```

---

google_elevation *Google elevation*

Description

The Google Maps Elevation API provides elevation data for all locations on the surface of the earth, including depth locations on the ocean floor (which return negative values).

Usage

```
google_elevation(
  df_locations = NULL,
  polyline = NULL,
  location_type = c("individual", "path"),
  samples = NULL,
  key = get_api_key("elevation"),
  simplify = TRUE,
  curl_proxy = NULL
)
```

Arguments

df_locations	data.frame of with two columns called 'lat' and 'lon' (or 'latitude' / 'longitude') used as the locations
polyline	string encoded polyline
location_type	string Specifies the results to be returned as individual locations or as a path. One of 'individual' or 'path'. If 'path', the data.frame df_locations must contain at least two rows. The order of the path is determined by the order of the rows.
samples	integer Required if location_type == "path". Specifies the number of sample points along a path for which to return elevation data. The samples parameter divides the given path into an ordered set of equidistant points along the path.
key	string A valid Google Developers Elevation API key
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object

Details

Locations can be specified as either a data.frame containing both a lat/latitude and lon/longitude column, or a single encoded polyline

Value

Either list or JSON string of the elevation data

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:

set_key("YOUR_GOOGLE_API_KEY")
## elevation data for the MCG in Melbourne
df <- data.frame(lat = -37.81659,
                 lon = 144.9841)

google_elevation(df_locations = df,
                 simplify = TRUE)
```

```
## elevation data from the MCG to the beach at Elwood (due south)
df <- data.frame(lat = c(-37.81659, -37.88950),
                 lon = c(144.9841, 144.9841))

df <- google_elevation(df_locations = df,
                       location_type = "path",
                       samples = 20,
                       simplify = TRUE)

## plot results
library(ggplot2)
df_plot <- data.frame(elevation = df$results$elevation,
                       location = as.integer(rownames(df$results)))

ggplot(data = df_plot, aes(x = location, y = elevation)) +
 geom_line()

## End(Not run)
```

`google_find_place` *Google Find Place*

Description

A Find Place request takes a text input, and returns a place. The text input can be any kind of Places data, for example, a name, address, or phone number

Usage

```
google_find_place(
  input,
  inputtype = c("textquery", "phonenumber"),
  language = NULL,
  fields = place_fields(),
  point = NULL,
  circle = NULL,
  rectangle = NULL,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("find_place")
)
```

Arguments

`input`	The text input specifying which place to search for (for example, a name, address, or phone number).

inputtype	The type of input. This can be one of either textquery or phonenumber. Phone numbers must be in international format (prefixed by a plus sign ("+"), followed by the country code, then the phone number itself).
language	string The language code, indicating in which language the results should be returned, if possible. Searches are also biased to the selected language; results in the selected language may be given a higher ranking. See the list of supported languages and their codes https://developers.google.com/maps/faq#languagesupport.
fields	vector of place data types to return. All Basic fields are returned by default. See details
point	vector of lat & lon values. Prefer results near this point.
circle	list of two elements, point (vector of lat & lon) and radius. Prefer results in this circle. Ignored if point is supplied
rectangle	list of two elements, sw (vector of lat & lon) and ne (vector of lat & lon) specifying the south-west and north-east bounds of a rectangle. Prefer results in this rectangle. Ignored if either point or circle are supplied
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string into a list.
curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key.

Details

Fields correspond to place search results https://developers.google.com/maps/documentation/places/web-service/search and are divided into three billing categories: Basic, Contact and Atmosphere.

Basic fields are billed at base rate, and incur no additional charges. Contact and atmosphere fields are billed at a hihger rate. See pricing sheet for more information https://mapsplatform.google.com/pricing/

- Basic - formatted_address, geometry, icon, id, name, permanently_closed, photos, place_id, plus_code, types
- Contact - opening_hours
- Atmosphere - price_level, rating

See Also

google_place_details google_places

Examples

```
## specifying fields

set_key( "your_api_key" )
```

```
google_find_place(
  input = "Museum of Contemporary Art Australia"
  , fields = c("photos","formatted_address","name","rating","opening_hours","geometry")
)

## Using location bias - circle
google_find_place(
  input = "Mongolian Grill"
  , circle = list(point = c(47.7, -122.2), radius = 2000)
)

## finding by a phone number
google_find_place(
 input = "+61293744000"
 , inputtype = "phonenumber"
)
```

---

google_geocode *Google geocoding*

Description

Geocoding is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers on a map, or position the map.

Usage

```
google_geocode(
  address,
  bounds = NULL,
  key = get_api_key("geocode"),
  language = NULL,
  region = NULL,
  components = NULL,
  simplify = TRUE,
  curl_proxy = NULL
)
```

Arguments

address	`string`. The street address that you want to geocode, in the format used by the national postal service of the country concerned
bounds	list of two, each element is a vector of lat/lon coordinates representing the south-west and north-east bounding box

key	string. A valid Google Developers Geocode API key
language	string. Specifies the language in which to return the results. See the list of supported languages: https://developers.google.com/maps/faq#using-google-maps-apis. If no langauge is supplied, the service will attempt to use the language of the domain from which the request was sent
region	string. Specifies the region code, specified as a ccTLD ("top-level domain"). See region basing for details https://developers.google.com/maps/documentation/directions/overview#RegionBiasing
components	data.frame of two columns, component and value. Restricts the results to a specific area. One or more of "route","locality","administrative_area", "postal_code","country"
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object

Value

Either list or JSON string of the geocoded address

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:

set_key("YOUR_GOOGLE_API_KEY")
df <- google_geocode(address = "MCG, Melbourne, Australia",
                     simplify = TRUE)

df$results$geometry$location
        lat      lng
1 -37.81659 144.9841

## using bounds
bounds <- list(c(34.172684,-118.604794),
               c(34.236144,-118.500938))

js <- google_geocode(address = "Winnetka",
                     bounds = bounds,
                     simplify = FALSE)

## using components
```

```
components <- data.frame(component = c("postal_code", "country"),
                         value = c("3000", "AU"))

df <- google_geocode(address = "Flinders Street Station",
                     components = components,
                     simplify = FALSE)


## End(Not run)
```

---

google_keys *Google Keys*

Description

Retrieves the list of Google Keys that have been set.

Usage

```
google_keys()
```

---

google_map *Google map*

Description

Generates a google map object

Usage

```
google_map(
  data = NULL,
  key = get_api_key("map"),
  location = NULL,
  zoom = NULL,
  min_zoom = NULL,
  max_zoom = NULL,
  map_bounds = c(-180, -90, 180, 90),
  width = NULL,
  height = NULL,
  padding = 0,
  styles = NULL,
  search_box = FALSE,
  update_map_view = TRUE,
  zoom_control = TRUE,
  map_type = c("roadmap", "satellite", "hybrid", "terrain"),
```

```
  map_type_control = TRUE,
  scale_control = FALSE,
  street_view_control = TRUE,
  rotate_control = TRUE,
  fullscreen_control = TRUE,
  libraries = NULL,
  split_view = NULL,
  split_view_options = NULL,
  geolocation = FALSE,
  event_return_type = c("list", "json")
)
```

Arguments

data	data to be used on the map. Either a data.frame, or an sf object. See details
key	A valid Google Maps API key.
location	numeric vector of latitude/longitude (in that order) coordinates for the initial starting position of the map. The map will automatically set the location and zoom if data is added through one of the various add_ functions. If null, the map will default to Melbourne, Australia.
zoom	integer representing the zoom level of the map (0 is fully zoomed out)
min_zoom	the maximum zoom level which will be displayed on the map
max_zoom	the minimum zoom level which will be displayed on the map
map_bounds	the visible bounds of the map, specified as a vector of four values of the form (xmin, ymin, xmax, ymax) (i.e., the form of the bounding box of 'sf' objects).
width	the width of the map
height	the height of the map
padding	the padding of the map
styles	JSON string representation of a valid Google Maps styles Array. See the Google documentation for details https://developers.google.com/maps/documentation/cloud-customization/cloud-based-map-styling
search_box	boolean indicating if a search box should be placed on the map
update_map_view	logical indicating if the map should center on the searched location
zoom_control	logical indicating if the zoom control should be displayed
map_type	defines the type of map to display. One of 'roadmap', 'satellite', 'terrain' or 'hybrid'
map_type_control	logical indicating if the map type control should be displayed
scale_control	logical indicating if the scale control should be displayed
street_view_control	logical indicating if the street view control should be displayed
rotate_control	logical indicating if the rotate control should be displayed

`fullscreen_control`
: logical indicating if the full screen control should be displayed

`libraries`
: vector containing the libraries you want to load. See details

`split_view`
: string giving the name of a UI output element in which to place a streetview representation of the map. Will only work in an interactive environment (shiny).

`split_view_options`
: list of options to pass to the split street view. valid list elements are `heading` and `pitch` see google_mapOutput

`geolocation`
: logical indicating if you want geolocation enabled

`event_return_type`
: the type of data to return to R from an interactive environment (shiny), either an R list, or raw json string.

Details

In order to use Google Maps you need a valid Google Maps Web JavaScript API key. See the Google Maps API documentation https://mapsplatform.google.com/

The data argument is only needed if you call other functions to add layers to the map, such as `add_markers()` or `add_polylines`. However, the data argument can also be passed into those functions as well.

The data can either be a data.frame containing longitude and latitude columns or an encoded polyline for plotting polylines and polygons, or an `sf` object.

The supported `sf` object types are

- POINT
- MULTIPOINT
- LINESTRING
- MULTILINESTRING
- POLYGON
- MULTIPOLYGON
- GEOMETRY

The libraries argument can be used to turn-off certain libraries from being called. By default the map will load

- visualization - includes the HeatmapLayer for visualising heatmaps https://developers.google.com/maps/documentation/javascript/visualization
- geometry - utility functions for computation of geometric data on the surface of the earth, including plotting encoded polylines. https://developers.google.com/maps/documentation/javascript/geometry
- places - enables searching for places. https://developers.google.com/maps/documentation/javascript/places
- drawing - provides a graphical interface for users to draw polygons, rectangles, circles and markers on the map. https://developers.google.com/maps/documentation/javascript/drawinglayer

See Also

google_mapOutput

Examples

```
## Not run:

map_key <- "your_api_key"

google_map(key = map_key, data = tram_stops) %>%
 add_markers() %>%
 add_traffic()

## style map using 'cobalt simplified' style
style <- '[{"featureType":"all","elementType":"all","stylers":[{"invert_lightness":true},
{"saturation":10},{"lightness":30},{"gamma":0.5},{"hue":"#435158"}]},
{"featureType":"road.arterial","elementType":"all","stylers":[{"visibility":"simplified"}]},
{"featureType":"transit.station","elementType":"labels.text","stylers":[{"visibility":"off"}]}]'

google_map(key = map_key, styles = style)


## End(Not run)
```

`google_map-shiny` *Shiny bindings for google_map*

Description

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

Usage

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

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

Arguments

`outputId`	output variable to read from
`width, height`	Must be a valid CSS unit (like `'100%'`, `'400px'`, `'auto'`) or a number, which will be coerced to a string and have `'px'` appended.
`expr`	An expression that generates a google_map
`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

```
## Not run:
library(shiny)
library(googleway)

ui <- fluidPage(google_mapOutput("map"))

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

  api_key <- "your_api_key"

  output$map <- renderGoogle_map({
    google_map(key = api_key)
  })
}

shinyApp(ui, server)

## using split view

library(shinydashboard)
library(googleway)

ui <- dashboardPage(

  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody(
    box(width = 6,
        google_mapOutput(outputId = "map")
    ),
    box(width = 6,
        google_mapOutput(outputId = "pano")
    )
  )
)

server <- function(input, output) {
  #set_key("your_api_key")

  output$map <- renderGoogle_map({
    google_map(location = c(-37.817386, 144.967463),
                zoom = 10,
                split_view = "pano")
  })
}

shinyApp(ui, server)


## End(Not run)
```

google_map_directions *Google Map Directions*

Description

Opens Google Maps in a browser with the resutls of the specified directions query

Usage

```
google_map_directions(
  origin = NULL,
  origin_place_id = NULL,
  destination = NULL,
  destination_place_id = NULL,
  travel_mode = NULL,
  dir_action = NULL,
  waypoints = NULL,
  waypoint_place_ids = NULL
)
```

Arguments

origin	string of an address or search term, or vector of lat/lon coordinates
origin_place_id	a Google place id (https://developers.google.com/maps/documentation/places/web-service/place-id). If used, you must also specify an origin
destination	string of an address or vector of lat/lon coordinates
destination_place_id	a Google place id (https://developers.google.com/maps/documentation/places/web-service/place-id). If used, you must also specify an destination
travel_mode	one of driving, walking, bicycling or transit. If not supplied, the Google Map will show one or more of the most relevant modes for the route.
dir_action	can only be "navigate". If set, the map will attempt to launch turn-by-turn navigation or route preview to the destination.
waypoints	List of either place names, addresses, or vectors of lat/lon coordinates. Up to 3 are allowed on mobile devices, and up to 9 otherwise.
waypoint_place_ids	vector of place_ids to match against the list of waypoints. If used, the waypoints must also be used.

Note

There is no need for an api key

Waypoints are not supported on all Google Map products. In those cases, this parameter will be ignored.

Examples

```
## Not run:

google_map_directions(origin = "Google Pyrmont NSW",
  destination = "QVB, Sydney", destination_place_id = "ChIJISz8NjyuEmsRFTQ9Iw7Ear8",
  travel_mode = "walking")


google_map_directions(origin = "Melbourne Cricket Ground",
  destination = "Flinders Street Station",
  dir_action = "navigate")

google_map_directions(origin = "Melbourne Cricket Ground",
  destination = "Flinders Street Station",
  travel_mode = "walking",
  waypoints = list("National Gallery of Victoria", c(-37.820860, 144.961894)))


google_map_directions(origin = "Paris, France",
  destination = "Cherbourg, France",
  travel_mode = "driving",
  waypoints = list("Versailles, France", "Chartres, France", "Le Mans, France",
    "Caen, France"))


google_map_directions(origin = "Paris, France",
  destination = "Cherbourg, France",
  travel_mode = "driving",
  waypoints = list("Versailles, France", "Chartres, France", "Le Mans, France",
    "Caen, France"),
  waypoint_place_ids = list("ChIJdUyx15R95kcRj85ZX8H8OAU",
 "ChIJKzGHdEgM5EcR_OBTT3nQoEA", "ChIJG2LvQNCI4kcRKXNoAsPi1Mc", "ChIJ06tnGbxCCkgRsfNjEQMwUsc"))


## End(Not run)
```

`google_map_panorama` *Google Map Panorama*

Description

Opens an interactive street view panorama in a browser

Usage

```
google_map_panorama(
  viewpoint = NULL,
  pano = NULL,
```

```
  heading = NULL,
  pitch = 0,
  fov = 90
)
```

Arguments

viewpoint	vector of lat/lon coordinates. If NULL, pano must be used.
pano	string of a specific panorama ID (see https://developers.google.com/maps/documentation/urls/get-started#pano-id). If NULL, viewpoint must be used.
heading	number between -180 and 360. Indicates the compass heading of the camera in degrees clockwise from north.
pitch	number between -90 and 90, specifying the angle, up or down, of the camera
fov	number between 10 and 100, determines the orizontal field of view of the image.

Examples

```
## Not run:

google_map_panorama(viewpoint = c(48.857832, 2.295226))

google_map_panorama(viewpoint = c(48.857832,2.295226),
  heading = -90, pitch = 38, fov = 80)

google_map_panorama(pano = "4U-oRQCNsC6u7r8gp02sLA")



## End(Not run)
```

---

google_map_search *Google Map Search*

Description

Opens a Google Map in a browser with the result of the specified search query.

Usage

```
google_map_search(query, place_id = NULL)
```

Arguments

query	string or vector of lat/lon coordinates (in that order)
place_id	a Google place id (https://developers.google.com/maps/documentation/places/web-service/place-id).

Details

If both parameters are given, the `query` is only used if Google Maps cannot find the `place_id`.

Note

There is no need for an api key

Examples

```
## Not run:

google_map_search("Melbourne, Victoria")

google_map_search("Restaruants")

## Melbourne Cricket Ground
google_map_search(c(-37.81997, 144.9834), place_id = "ChIJgWIaV5VC1moR-bKgR9ZfV2M")

## Without the place_id, no additional place inforamtion is displayed on the map
google_map_search(c(-37.81997, 144.9834))


## End(Not run)
```

---

google_map_update *Google map update*

Description

Update a Google map in a shiny app. Use this function whenever the map needs to respond to reactive content.

Usage

```
google_map_update(
  map_id,
  session = shiny::getDefaultReactiveDomain(),
  data = NULL,
  deferUntilFlush = TRUE
)
```

Arguments

`map_id`	string containing the output ID of the map in a shiny application.
`session`	the Shiny session object to which the map belongs; usually the default value will suffice.
`data`	data to be used in the map. See the details section for `google_map`.

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.

Examples

```
## Not run:

library(shiny)
library(googleway)

ui <- pageWithSidebar(
  headerPanel("Toggle markers"),
  sidebarPanel(
    actionButton(inputId = "markers", label = "toggle markers")
  ),
  mainPanel(
    google_mapOutput("map")
  )
)

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

  # api_key <- "your_api_key"

  df <- structure(list(lat = c(-37.8201904296875, -37.8197288513184,
  -37.8191299438477, -37.8187675476074, -37.8186187744141, -37.8181076049805
  ), lon = c(144.968612670898, 144.968414306641, 144.968139648438,
 144.967971801758, 144.967864990234, 144.967636108398), weight = c(31.5698964400217,
 97.1629025738221, 58.9051092562731, 76.3215389118996, 37.8982300488278,
 77.1501972114202), opacity = c(0.2, 0.2, 0.2, 0.2, 0.2, 0.2)), .Names = c("lat",
 "lon", "weight", "opacity"), row.names = 379:384, class = "data.frame")


  output$map <- renderGoogle_map({
    google_map(key = api_key)
  })

  observeEvent(input$markers,{

    if(input$markers %% 2 == 1){
      google_map_update(map_id = "map") %>%
        add_markers(data = df)
    }else{
      google_map_update(map_id = "map") %>%
        clear_markers()
    }
  })
 }
shinyApp(ui, server)

## End(Not run)
```

google_map_url *Google Map Url*

Description

Opens a Google Map in a browser

Usage

```
google_map_url(
  center = NULL,
  zoom = 15,
  basemap = c("roadmap", "satellite", "hybrid", "terrain"),
  layer = c("none", "transit", "traffic", "bicycling")
)
```

Arguments

center	vector of lat/lon coordinates which defines the centre of the map window
zoom	number that sets the zoom level of the map (from 0 to 21)
basemap	defines the typ eof map to display.
layer	defines an extra layer to display on the map, if any.

Examples

```
## Not run:

google_map_url()

google_map_url(center = c(-37.817727, 144.968246))

google_map_url(center = c(-37.817727, 144.968246), zoom = 5)

google_map_url(center = c(-37.817727, 144.968246), basemap = "terrain")

google_map_url(center = c(-37.817727, 144.968246), layer = "traffic")


## End(Not run)
```

google_map_view *google map view*

Description

google map view

Usage

```
google_map_view(map, location, zoom)
```

Arguments

map	a googleway map object created from google_map()
location	numeric vector of latitude/longitude (in that order) coordinates for the initial starting position of the map. The map will automatically set the location and zoom if data is added through one of the various add_ functions. If null, the map will default to Melbourne, Australia.
zoom	integer representing the zoom level of the map (0 is fully zoomed out)

google_nearestRoads *Nearest Roads*

Description

Takes up to 100 independent coordinates and returns the closest road segment for each point. The points passed do not need to be part of a continuous path.

Usage

```
google_nearestRoads(
  df_points,
  lat = NULL,
  lon = NULL,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("roads")
)
```

Arguments

df_points	data.frame with at least two columns specifying the latitude & longitude coordinates, with a maximum of 100 pairs of coordinates.
lat	string specifying the column of df_path containing the 'latitude' coordinates. If left NULL, a best-guess will be made
lon	string specifying the column of df_path containing the 'longitude' coordinates. If left NULL, a best-guess will be made
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key

See Also

google_snapToRoads

Examples

```
## Not run:

key <- 'your_api_key'

df_points <- read.table(text = "lat lon
 60.1707 24.9426
 60.1708 24.9424
 60.1709 24.9423", header = T)

google_nearestRoads(df_points, key = key)


## End(Not run)
```

---

google_places *Google places*

Description

The Google Places API Web Service allows you to query for place information on a variety of categories, such as: establishments, prominent points of interest, geographic locations, and more.

Usage

```
google_places(
  search_string = NULL,
  location = NULL,
  radius = NULL,
```

```
  rankby = NULL,
  keyword = NULL,
  language = NULL,
  name = NULL,
  place_type = NULL,
  price_range = NULL,
  open_now = NULL,
  page_token = NULL,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("places"),
  radar = NULL
)
```

Arguments

search_string	string A search term representing a place for which to search. If blank, the location argument must be used.
location	numeric vector of latitude/longitude coordinates (in that order) around which to retrieve place information.
radius	numeric Defines the distance (in meters) within which to return place results. Required if only a location search is specified. The maximum allowed radius is 50,000 meters. Radius must not be included if rankby is used. see Details.
rankby	string Specifies the order in which results are listed. Possible values are "prominence" or "distance". If rankby = distance, then one of keyword, name or place_type must be specified. If a search_string is used then rankby is ignored.
keyword	string A term to be matched against all content that Google has indexed for this place, including but not limited to name, type, and address, as well as customer reviews and other third-party content.
language	string The language code, indicating in which language the results should be returned, if possible. Searches are also biased to the selected language; results in the selected language may be given a higher ranking. See the list of supported languages and their codes https://developers.google.com/maps/faq#languagesupport.
name	string vector One or more terms to be matched against the names of places. Ignored when used with a search_string. Results will be restricted to those containing the passed name values. Note that a place may have additional names associated with it, beyond its listed name. The API will try to match the passed name value against all of these names. As a result, places may be returned in the results whose listed names do not match the search term, but whose associated names do.
place_type	string Restricts the results to places matching the specified type. Only one type may be specified. For a list of valid types, please visit https://developers.google.com/maps/documentation/places/web-service/supported_types.
price_range	numeric vector Specifying the minimum and maximum price ranges. Values range between 0 (most affordable) and 4 (most expensive).

open_now	logical Returns only those places that are open for business at the time the query is sent. Places that do not specify opening hours in the Google Places database will not be returned if you include this parameter in your query.
page_token	string Returns the next 20 results from a previously run search. Setting a page_token parameter will execute a search with the same parameters used in a previous search. All parameters other than page_token will be ignored. The page_token can be found in the result set of a previously run query.
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string into a list.
curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key.
radar	deprecated, no longer used

Details

A Nearby Search (using google_places) lets you search for places within a specified area. You can refine your search request by supplying keywords or specifying the type of place you are searching for.

With the Places service you can perform three kinds of searches:

- Nearby Search
- Text Search
- Place Details request

A Nearby search lets you search for places within a specified area or by keyword. A Nearby search must always include a location, which can be specified as a point defined by a pair of lat/lon coordinates, or a circle defined by a point and a radius.

A Text search returns information about a set of places based on the search_string. The service responds with a list of places matching the string and any location bias that has been set.

A Place Detail search (using google_place_details) can be performed when you have a given place_id from one of the other three search methods.

radius - Required when only using a location search, radius defines the distance (in meters) within which to return place results. The maximum allowed radius is 50,000 meters. Note that radius must not be included if rankby = distance is specified.

radius - Optional when using a search_string. Defines the distance (in meters) within which to bias place results. The maximum allowed radius is 50,000 meters. Results inside of this region will be ranked higher than results outside of the search circle; however, prominent results from outside of the search radius may be included.

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Note

The Google Places API Web Service enforces a default limit of 1,000 free requests per 24 hour period, calculated as the sum of client-side and server-side requets. See https://developers.google.com/maps/documentation/places/web-service/usage-and-billing for details.

Use of the Places Library must be in accordance with the polices described for the Google Places API Web Service https://developers.google.com/maps/documentation/places/web-service/policies

See Also

google_place_details google_find_place

Examples

```
## Not run:

## query restaurants in Melbourne (will return 20 results)
api_key <- 'your_api_key'

res <- google_places(search_string = "Restaurants in Melbourne, Australia",
                     key = api_key)

## use the 'next_page_token' from the previous search to get the next 20 results
res_next <- google_places(search_string = "Restaurants in Melbourne, Australia",
                          page_token = res$next_page_token,
                          key = api_key)

## search for a specific place type
google_places(location = c(-37.817839,144.9673254),
              place_type = "bicycle_store",
              radius = 20000,
              key = api_key)

## search for places that are open at the time of query
 google_places(search_string = "Bicycle shop, Melbourne, Australia",
               open_now = TRUE,
               key = api_key)


## End(Not run)
```

---

google_place_autocomplete

Google place autocomplete

Description

The Place Autocomplete service is a web service that returns place predictions in response to an HTTP request. The request specifies a textual search string and optional geographic bounds. The service can be used to provide autocomplete functionality for text-based geographic searches, by returning places such as businesses, addresses and points of interest as a user types.

Usage

```
google_place_autocomplete(
  place_input,
  location = NULL,
  radius = NULL,
  language = NULL,
  place_type = NULL,
  components = NULL,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("place_autocomplete")
)
```

Arguments

place_input	string The text string on which to search. The Place Autocomplete service will return candidate matches based on this string and order results based on their perceived relevance.
location	numeric vector of latitude/longitude coordinates (in that order) the point around which you wish to retrieve place information
radius	numeric The distance (in meters) within which to return place results. Note that setting a radius biases results to the indicated area, but may not fully restrict results to the specified area
language	string The language code, indicating in which language the results should be returned, if possible. Searches are also biased to the selected language; results in the selected language may be given a higher ranking. See the list of supported languages and their codes https://developers.google.com/maps/faq#languagesupport
place_type	string Restricts the results to places matching the specified type. Only one type may be specified (if more than one type is provided, all types following the first entry are ignored). For a list of valid types, please visit https://developers.google.com/maps/documentation/places/web-service/autocomplete
components	string of length 1 which identifies a grouping of places to which you would like to restrict your results. Currently, you can use components to filter by country only. The country must be passed as a two character, ISO 3166-1 Alpha-2 compatible country code. For example: components=country:fr would restrict your results to places within France.
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string

curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key

Examples

```
## Not run:

## search for 'Maha' Restaurant, Melbourne
google_place_autocomplete("Maha Restaurant", key = key)

## search for 'Maha' Restaurant, exclusively in Australia
google_place_autocomplete("maha Restaurant", component = "au", key = key)


## End(Not run)
```

---

google_place_details *Google place details*

Description

Once you have a place_id from a Place Search, you can request more details about a particular establishment or point of interest by initiating a Place Details request. A Place Details request returns more comprehensive information about the indicated place such as its complete address, phone number, user rating and reviews.

Usage

```
google_place_details(
  place_id,
  language = NULL,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("place_details")
)
```

Arguments

place_id	string A textual identifier that uniquely identifies a place, usually of the form ChIJrTLr-GyuEmsRBfy61i59si0, returned from a place search
language	string The language code, indicating in which language the results should be returned, if possible. Searches are also biased to the selected language; results in the selected language may be given a higher ranking. See the list of supported languages and their codes https://developers.google.com/maps/faq#languagesupport
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

See Also

google_places

Examples

```
## Not run:
## search for a specific restaurant, Maha, in Melbourne, firstly using google_places()
res <- google_places(search_string = "Maha Restaurant, Melbourne, Australia",
                     radius = 1000,
                     key = key)

## request more details about the restaurant using google_place_details()
google_place_details(place_id = res$results$place_id, key = key)


## End(Not run)
```

---

google_reverse_geocode

Google reverse geocoding

Description

Reverse geocoding is the process of converting geographic coordinates into a human-readable address.

Usage

```
google_reverse_geocode(
  location,
  result_type = NULL,
  location_type = NULL,
  language = NULL,
  key = get_api_key("reverse_geocode"),
  simplify = TRUE,
  curl_proxy = NULL
)
```

Arguments

location	numeric vector of lat/lon coordinates.
result_type	string vector - one or more address types. See https://developers.google.com/maps/documentation/geocoding/overview#Types for list of available types.
location_type	string vector specifying a location type will restrict the results to this type. If multiple types are specified, the API will return all addresses that match any of the types
language	string specifies the language in which to return the results. See the list of supported languages: https://developers.google.com/maps/faq#using-google-maps-apis. If no langauge is supplied, the service will attempt to use the language of the domain from which the request was sent
key	string. A valid Google Developers Geocode API key
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object

Value

Either list or JSON string of the geocoded address

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:
## searching for the street address for the rooftop location type
google_reverse_geocode(location = c(-37.81659, 144.9841),
                       result_type = c("street_address"),
                       location_type = "rooftop",
                       key = "<your valid api key>")

## End(Not run)
```

`google_snapToRoads` *Snap To Roads*

Description

Takes up to 100 GPS coordinates collected along a route and returns a similar set of data, with the points snapped to the most likely roads the vehicle was treveling along

Usage

```
google_snapToRoads(
  df_path,
  lat = NULL,
  lon = NULL,
  interpolate = FALSE,
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("roads")
)
```

Arguments

`df_path`	`data.frame` with at least two columns specifying the latitude & longitude coordinates, with a maximum of 100 pairs of coordinates.
`lat`	string specifying the column of `df_path` containing the 'latitude' coordinates. If left NULL, a best-guess will be made.
`lon`	string specifying the column of `df_path` containing the ' longitude' coordinates. If left NULL, a best-guess will be made.
`interpolate`	logical indicating whether to interpolate a path to include all points forming the full road-geometry. When `TRUE`, additional interpolated points will also be returned, resulting in a path that smoothly follows the geometry of the road, even around corners and through tunnels. Interpolated paths will most likely contain more ponts that the original path.
`simplify`	`logical` - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
`curl_proxy`	a curl proxy object
`key`	`string` A valid Google Developers Places API key

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Note

The snapping algorithm works best for points that are not too far apart. If you observe odd snapping behaviour, try creating paths that have points closer together. To ensure the best snap-to-road quality, you should aim to provide paths on which consecutive pairs of points are within 300m of each other. This will also help in handling any isolated, long jumps between consecutive points caused by GPS signal loss or noise.

See Also

google_nearestRoads

Examples

```
## Not run:

key <- 'your_api_key'

df_path <- read.table(text = "lat lon
-35.27801 149.12958
-35.28032 149.12907
-35.28099 149.12929
-35.28144 149.12984
-35.28194 149.13003
-35.28282 149.12956
-35.28302 149.12881
-35.28473 149.12836", header = T)

google_snapToRoads(df_path, key = key, interpolate = TRUE, simplify = TRUE)



## End(Not run)
```

---

google_speedLimits *Speed Limits*

Description

Returns the posted speed limit for a given road segment. In the case of road segments with variable speed limits, the default speed limit for the segment is returned. The speed limits service is only available to Google Maps API Premium Plan customers with an Asset Tracking license.

Usage

```
google_speedLimits(
  df_path = NULL,
  lat = NULL,
  lon = NULL,
```

```
  placeIds = NULL,
  units = c("KPH", "MPH"),
  simplify = TRUE,
  curl_proxy = NULL,
  key = get_api_key("roads")
)
```

Arguments

df_path	data.frame with at least two columns specifying the latitude & longitude coordinates, with a maximum of 100 pairs of coordinates.
lat	string specifying the latitude column
lon	string specifying the longitude column
placeIds	vector of Place IDs of the road segments. Place IDs are returned in response to google_snapToRoads and google_nearestRoads reqeusts. You can pass up to 100 placeIds at a time
units	Whether to return speed limits in kilometers or miles per hour
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object
key	string A valid Google Developers Places API key

Note

The accuracy of speed limit data returned by Google Maps Roads API can not be guaranteed. The speed limit data provided is not real-time, and may be estimated, inaccurate, incomplete, and / or outdated.

---

google_streetview *Google street view*

Description

Displays a static street view image from Google Maps Street View Image API

Usage

```
google_streetview(
  location = NULL,
  panorama_id = NULL,
  size = c(400, 400),
  heading = NULL,
  fov = 90,
  pitch = 0,
  output = c("plot", "html"),
```

```
  response_check = FALSE,
  signature = NULL,
  key = get_api_key("streetview")
)
```

Arguments

location	numeric vector of lat/lon coordinates, or an address string.
panorama_id	a specific panorama ID.
size	numeric vector of length 2, specifying the output size of the image in pixels, given in width x height. For example, c(600, 400) returns an image 600 pixles wide and 400 pixles high.
heading	indicates the compass heading of the camera. Accepted values are from 0 to 360 (both 0 and 360 indicate north), 90 indicates east, 180 south and 270 west. If no heading is specified a value will be calculated that directs the camera to wards the specified location, from the point at which the closest photograph was taken.
fov	determines the horizontal field of view of the image. The field of view is expressed in degrees, with a maximum allowed value of 120. When dealing with a fixed-size viewport, as with Street View image of a set size, field of view in essence represents zoom, with small numbers indicating a higher level of zoom
pitch	specifies the up or down angle of the camera relative to the Street View vehicle. This is often, but not always, flat horizontal. Positive values angle the camera up (with 90 degrees indicating straight up); negative values angle the camera down (with -90 indicating straight down)
output	specifies whether the result should be displayed in R's viewer, or embedded as HTML inside a webpage.
response_check	logical indicating if the function should first check if the image is available. If TRUE and no image is available, a warning message is printed and no image will be downloaded. if FALSE and no image is available, a blank image will be displayed saying 'Sorry, we have no imagery here'.
signature	a digitial signature used to verify that any site generating requests using your API key is authorised to do so. See Google Documentation for further details https://developers.google.com/maps/documentation/streetview/overview
key	string. A valid Google Developers Street View Image API key

Examples

```
## Not run:

## download and display an image
# key <- "your_api_key"
google_streetview(location = c(-37.817714, 144.96726),
    size = c(400,400), output = "plot",
    key = key)
```

```
## no response check - display 'sorry' message
google_streetview(location = c(-37.8, 144),
   size = c(400,400),
   panorama_id = NULL,
   output = "plot",
   heading = 90,
   fov = 90,
   pitch = 0,
   response_check = FALSE,
   key = key)


## embed an image of Flinders Street Station into a Shiny webpage
library(shiny)
library(googleway)

ui <- fluidPage(
  uiOutput(outputId = "myStreetview")
)

server <- function(input, output){
  key <- "your_api_key"

  output$myStreetview <- renderUI({
    tags$img(src = google_streetview(location = c(-37.817714, 144.96726),
                                     size = c(400,400), output = "html",
                                     key = key),  width = "400px", height = "400px")
  })
}

shinyApp(ui, server)


## End(Not run)
```

---

google_timezone *Google timezone*

Description

The Google Maps Time Zone API provides time offset data for locations on the surface of the earth. You request the time zone information for a specific latitude/longitude pair and date.

Usage

```
google_timezone(
  location,
  timestamp = Sys.time(),
  language = NULL,
  simplify = TRUE,
```

```
  curl_proxy = NULL,
  key = get_api_key("timezone")
)
```

Arguments

location	vector of lat/lon pair
timestamp	POSIXct The Google Maps Time Zone API uses the timestamp to determine whether or not Daylight Savings should be applied. Will default to the current system time.
language	string specifies the language in which to return the results. See the list of supported languages: https://developers.google.com/maps/faq#using-google-maps-apis. If no langauge is supplied, the service will attempt to use the language of the domain from which the request was sent.
simplify	logical - TRUE indicates the returned JSON will be coerced into a list. FALSE indicates the returend JSON will be returned as a string
curl_proxy	a curl proxy object
key	string A valid Google Developers Timezone API key.

Value

Either list or JSON string of the timezone

API use and limits

The amount of queries you can make to Google's APIs is dependent on both the service and the API you are using.

Each API has specific quotas and limits. Check Google's API documentation for details.

View your usage at the Google Cloud Console https://console.cloud.google.com/

Each API can only accept and return one request at a time. If you write a loop to make multiple API calls you should ensure you don't go over your quota / limits during the loop.

Examples

```
## Not run:
google_timezone(location = c(-37.81659, 144.9841),
                timestamp = as.POSIXct("2016-06-05"),
                key = "<your valid api key>")

## End(Not run)
```

map_styles *Map Styles*

Description

Various styles for a `google_map()` map.

Usage

```
map_styles()
```

Value

list of styles

Note

you can generate your own map styles at https://mapstyle.withgoogle.com/

Examples

```
## Not run:
map_key <- "your_map_key"
google_map(key = map_key, style = map_styles()$silver)


## End(Not run)
```

melbourne *Melbourne*

Description

Polygons for Melbourne and the surrounding area

Usage

```
melbourne
```

Format

A data frame with 397 observations and 7 variables

polygonId a unique identifier for each polygon

pathId an identifier for each path that define a polygon

SA2_NAME statistical area 2 name of the polygon

SA3_NAME statistical area 3 name of the polygon

SA4_NAME statistical area 4 name of the polygon

AREASQKM area of the SA2 polygon

polyline encoded polyline that defines each `pathId`

Details

This data set is a subset of the Statistical Area Level 2 (SA2) ASGS Edition 2016 data released by the Australian Bureau of Statistics https://www.abs.gov.au

The data is realsed under a Creative Commons Attribution 2.5 Australia licence https://creativecommons.org/licenses/by/2.5/au/

`place_fields` *Place Fields*

Description

Convenience function to return all the valid basic field values for use in a google_find_place search

Usage

```
place_fields()
```

`set_key` *Set Key*

Description

Sets an API key so it's available for all API calls. See details

Usage

```
set_key(
  key,
 api = c("default", "map", "directions", "distance", "elevation", "geocode", "places",
   "find_place", "place_autocomplete", "places_details", "roads", "streetview",
    "timezone")
)
```

Arguments

key	Google API key
api	The api for which the key applies. If NULL, the api_key is assumed to apply to all APIs

Details

Use set_key to make API keys available for all the google_ functions, so you don't need to specify the key parameter within those functions (for example, see google_directions).

The api argument is useful if you use a different API key to access different APIs. If you just use one API key to access all the APIs, there is no need to specify the api parameter, the default value api_key will be used.

Examples

```
## not specifying 'api' will add the key to the 'api_key' list element
set_key(key = "xxx_your_api_key_xxx")

## api key for directions
set_key(key = "xxx_your_api_key_xxx", api = "directions")

## api key for maps
set_key(key = "xxx_your_api_key_xxx", api = "map")
```

---

tram_route *Tram Route*

Description

The latitude and longitude coordinates specifying the path tram 35 follows in Melbourne.

Usage

```
tram_route
```

Format

A data frame with 55 observations and 3 variables

shape_pt_lat the latitude of each point in the route

shape_pt_lon the longitude of each point in the route

shape_pt_sequence the position in the sequence of coordinates for each point

Details

The data is taken from the PTV GTFS data

tram_stops *Tram stops along tram route 35 in Melbourne*

Description

A data set containing the latitude and longitude coordinates of tram stops along route 35 in Melbourne.

Usage

```
tram_stops
```

Format

A data frame with 41 observations and 4 variables

stop_id unique ID for each stop

stop_name the name of each stop

stop_lat the latitude of the stop

stop_lon the longitude of the stop

Details

The data is taken from the PTV GTFS data

update_circles *Update circles*

Description

Updates specific colours and opacities of specified circles Designed to be used in a shiny application.

Usage

```
update_circles(
  map,
  data,
  id,
  radius = NULL,
  draggable = NULL,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  fill_colour = NULL,
```

```
  fill_opacity = NULL,
  info_window = NULL,
  layer_id = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
id	string representing the column of data containing the id values for the shapes. The id values must be present in the original data supplied to in order for the shape to be udpated.
radius	either a string specifying the column of data containing the radius of each circle, OR a numeric value specifying the radius of all the circles (radius is expressed in metres)
draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.

Note

Any circles (as specified by the `id` argument) that do not exist in the `data` passed into `add_circles()` will not be added to the map. This function will only update the circles that currently exist on the map when the function is called.

`update_geojson` *update geojson*

Description

Updates a geojson layer by a specified style. Designed to work within an interactive environment (e.g. shiny)

Usage

```
update_geojson(map, layer_id = NULL, style)
```

Arguments

`map`	a googleway map object created from `google_map()`
`layer_id`	single value specifying an id for the layer.
`style`	Style options for the geoJSON. See details

Details

The style object can either be a valid JSON string, or a named list. The style object will contain the following fields

- property : the property of the geoJSON that contains the `value`
- value : the value of the geoJSON that identifies the feature to be updated
- features : a list (or JSON object) of features to be updated

see add_geojson for valid features

Examples

```
## Not run:

style <- paste0('{
    "property" : "AREASQKM",
    "value" : 5,
    "operator" : ">=",
    "features" : {
      "fillColor" : "red",
      "strokeColor" : "red"
    }
  }')
```

```
google_map(key = map_key) %>%
    add_geojson(data = geo_melbourne) %>%
    update_geojson(style = style)

lst_style <- list(property = "AREASQKM", operator = "<=", value = 5,
   features = list(fillColor = "red",
   strokeColor = "red"))

google_map(key = map_key) %>%
    add_geojson(data = geo_melbourne) %>%
    update_geojson(style = lst_style)

## Styling a specific feature
style <- '{"property" : "SA2_NAME", "value" : "Abbotsford", "features" : { "fillColor" : "red" } }'
google_map(key = map_key) %>%
  add_geojson(data = geo_melbourne) %>%
  update_geojson(style = style)


## End(Not run)
```

---

update_heatmap *update heatmap*

Description

updates a heatmap layer

Usage

```
update_heatmap(
  map,
  data,
  lat = NULL,
  lon = NULL,
  weight = NULL,
  option_gradient = NULL,
  option_dissipating = FALSE,
  option_radius = 0.01,
  option_opacity = 0.6,
  layer_id = NULL,
  update_map_view = TRUE,
  digits = 4,
  legend = F,
  legend_options = NULL
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
lat	string specifying the column of data containing the 'latitude' coordinates. If left NULL, a best-guess will be made
lon	string specifying the column of data containing the 'longitude' coordinates. If left NULL, a best-guess will be made
weight	string specifying the column of data containing the 'weight' associated with each point. If NULL, each point will get a weight of 1.
option_gradient	vector of colours to use as the gradient colours. see Details
option_dissipating	logical Specifies whether heatmaps dissipate on zoom. When dissipating is FALSE the radius of influence increases with zoom level to ensure that the color intensity is preserved at any given geographic location. When set to TRUE you will likely need a greater option_radius value. Defaults to FALSE.
option_radius	numeric. The radius of influence for each data point, in pixels. Defaults to 0.01
option_opacity	The opacity of the heatmap, expressed as a number between 0 and 1. Defaults to 0.6.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
update_map_view	logical specifying if the map should re-centre according to the shapes
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.

Details

The option_gradient is only used to craete the legend, and not to change the colours of the heat layer. If you are not displaying a legend this argument is not needed. If you are displaying a legend, you should provide the same gardient as in the add_heatmap call.

Examples

```
## Not run:

map_key <- 'your_api_key'

set.seed(20170417)
df <- tram_route
df$weight <- sample(1:10, size = nrow(df), replace = T)
```

```
google_map(key = map_key, data = df) %>%
 add_heatmap(lat = "shape_pt_lat", lon = "shape_pt_lon", weight = "weight",
             option_radius = 0.001)

## update by adding the same data again to double the number of points at each location
df_update <- df
google_map(key = map_key, data = df) %>%
 add_heatmap(lat = "shape_pt_lat", lon = "shape_pt_lon", weight = "weight",
             option_radius = 0.001) %>%
 update_heatmap(df_update, lat = "shape_pt_lat", lon = "shape_pt_lon")


## End(Not run)
```

---

update_polygons *Update polygons*

Description

Updates specific colours and opacities of specified polygons. Designed to be used in a shiny application.

Usage

```
update_polygons(
  map,
  data,
  id,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  fill_colour = NULL,
  fill_opacity = NULL,
  info_window = NULL,
  layer_id = NULL,
  palette = NULL,
  legend = F,
  legend_options = NULL
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.

id	string representing the column of data containing the id values for the shapes. The id values must be present in the original data supplied to in order for the shape to be udpated.
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.

Note

Any polygons (as specified by the id argument) that do not exist in the data passed into add_polygons() will not be added to the map. This function will only update the polygons that currently exist on the map when the function is called.

Examples

```
## Not run:

map_key <- 'your_api_key'

pl_outer <- encode_pl(lat = c(25.774, 18.466,32.321),
                      lon = c(-80.190, -66.118, -64.757))

pl_inner <- encode_pl(lat = c(28.745, 29.570, 27.339),
                      lon = c(-70.579, -67.514, -66.668))

pl_other <- encode_pl(c(21,23,22), c(-50, -49, -51))

## using encoded polylines
```

```
df <- data.frame(id = c(1,1,2),
                 colour = c("#00FF00", "#00FF00", "#FFFF00"),
                 polyline = c(pl_outer, pl_inner, pl_other),
                 stringsAsFactors = FALSE)

google_map(key = map_key) %>%
  add_polygons(data = df, polyline = 'polyline', id = 'id', fill_colour = 'colour')

df_update <- df[, c("id", "colour")]
df_update$colour <- c("#FFFFFF", "#FFFFFF", "#000000")

google_map(key = map_key) %>%
  add_polygons(data = df, polyline = 'polyline', id = 'id', fill_colour = 'colour') %>%
  update_polygons(data = df_update, id = 'id', fill_colour = 'colour')


df <- aggregate(polyline ~ id + colour, data = df, list)

google_map(key = map_key) %>%
  add_polygons(data = df, polyline = 'polyline', fill_colour = 'colour')

google_map(key = map_key) %>%
  add_polygons(data = df, polyline = 'polyline', id = 'id', fill_colour = 'colour') %>%
  update_polygons(data = df_update, id = 'id', fill_colour = 'colour')


## using coordinates
df <- data.frame(id = c(rep(1, 6), rep(2, 3)),
                 lineId = c(rep(1, 3), rep(2, 3), rep(1, 3)),
                 lat = c(25.774, 18.466, 32.321, 28.745, 29.570, 27.339, 21, 23, 22),
             lon = c(-80.190, -66.118, -64.757, -70.579, -67.514, -66.668, -50, -49, -51))

google_map(key = map_key) %>%
  add_polygons(data = df, lat = 'lat', lon = 'lon', id = 'id', pathId = 'lineId')

google_map(key = map_key) %>%
  add_polygons(data = df, lat = 'lat', lon = 'lon', id = 'id', pathId = 'lineId') %>%
  update_polygons(data = df_update, id = 'id', fill_colour = 'colour')


## End(Not run)
```

`update_polylines` *Update polylines*

Description

Updates specific attributes of polylines. Designed to be used in a shiny application.

Usage

```
update_polylines(
  map,
  data,
  id,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  info_window = NULL,
  layer_id = NULL,
  palette = NULL,
  legend = F,
  legend_options = NULL
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
id	string representing the column of data containing the id values for the shapes. The id values must be present in the original data supplied to in order for the shape to be udpated.
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.

Note

Any polylines (as specified by the id argument) that do not exist in the data passed into add_polylines() will not be added to the map. This function will only update the polylines that currently exist on the map when the function is called.

Examples

```
## Not run:

map_key <- 'your_api_key'

## coordinate columns
## plot polylines using default attributes
df <- tram_route
df$id <- c(rep(1, 27), rep(2, 28))

df$colour <- c(rep("#00FFFF", 27), rep("#FF00FF", 28))

google_map(key = map_key) %>%
  add_polylines(data = df, lat = 'shape_pt_lat', lon = 'shape_pt_lon',
                stroke_colour = "colour", id = 'id')

## specify width and colour attributes to update
df_update <- data.frame(id = c(1,2),
                        width = c(3,10),
                        colour = c("#00FF00", "#DCAB00"))

google_map(key = map_key) %>%
  add_polylines(data = df, lat = 'shape_pt_lat', lon = 'shape_pt_lon',
                stroke_colour = "colour", id = 'id') %>%
  update_polylines(data = df_update, id = 'id', stroke_weight = "width",
                   stroke_colour = 'colour')


## encoded polylines
pl <- sapply(unique(df$id), function(x){
  encode_pl(lat = df[ df$id == x , 'shape_pt_lat'], lon = df[ df$id == x, 'shape_pt_lon'])
})

df <- data.frame(id = c(1, 2), polyline = pl)

df_update <- data.frame(id = c(1,2),
                        width = c(3,10),
                        var = c("a","b"))

google_map(key = map_key) %>%
  add_polylines(data = df, polyline = 'polyline')

google_map(key = map_key) %>%
  add_polylines(data = df, polyline = 'polyline') %>%
  update_polylines(data = df_update, id = 'id', stroke_weight = "width",
                   stroke_colour = 'var')


## End(Not run)
```

update_rectangles *Update rectangles*

Description

Updates specific colours and opacities of specified rectangles Designed to be used in a shiny application.

Usage

```
update_rectangles(
  map,
  data,
  id,
  draggable = NULL,
  stroke_colour = NULL,
  stroke_weight = NULL,
  stroke_opacity = NULL,
  fill_colour = NULL,
  fill_opacity = NULL,
  info_window = NULL,
  layer_id = NULL,
  digits = 4,
  palette = NULL,
  legend = F,
  legend_options = NULL
)
```

Arguments

map	a googleway map object created from google_map()
data	data frame containing the data to use in the layer. If Null, the data passed into google_map() will be used.
id	string representing the column of data containing the id values for the shapes. The id values must be present in the original data supplied to in order for the shape to be udpated.
draggable	string specifying the column of data defining if the polygon is 'draggable'. The column of data should be logical (either TRUE or FALSE)
stroke_colour	either a string specifying the column of data containing the stroke colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
stroke_weight	either a string specifying the column of data containing the stroke weight of each shape, or a number indicating the width of pixels in the line to be applied to all the shapes
stroke_opacity	either a string specifying the column of data containing the stroke opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes

fill_colour	either a string specifying the column of data containing the fill colour of each shape, or a valid hexadecimal numeric HTML style to be applied to all the shapes
fill_opacity	either a string specifying the column of data containing the fill opacity of each shape, or a value between 0 and 1 that will be applied to all the shapes
info_window	string specifying the column of data to display in an info window when a shape is clicked.
layer_id	single value specifying an id for the layer. Use this value to distinguish between shape layers for when using any update_ function, and for separating legends.
digits	integer. Use this parameter to specify how many digits (decimal places) should be used for the latitude / longitude coordinates.
palette	a function, or list of functions, that generates hex colours given a single number as an input. See details.
legend	either a logical indiciating if the legend(s) should be displayed, or a named list indicating which colour attributes should be included in the legend.
legend_options	A list of options for controlling the legend.

Note

Any rectangles (as specified by the id argument) that do not exist in the data passed into add_rectangles() will not be added to the map. This function will only update the rectangles that currently exist on the map when the function is called.

---

update_style *Update style*

Description

Updates the map with the given styles

Usage

```
update_style(map, styles = NULL)
```

Arguments

map	a googleway map object created from google_map()
styles	JSON string representation of a valid Google Maps styles Array. See the Google documentation for details https://developers.google.com/maps/documentation/cloud-customization/cloud-based-map-styling

Note

This function is intended for use with google_map_update in an interactive shiny environment. You can set the styles of the original map using the styles argument of google_map

`%>%` *Pipe*

Description

Uses the pipe operator (`%>%`) to chain statements. Useful for adding layers to a `google_map`

Arguments

`lhs`, `rhs`	A google map and a layer to add to it

Examples

```
## Not run:

key <- "your_api_key"
google_map(key = key) %>%
add_traffic()


## End(Not run)
```

Index