Package: toro (via r-universe)

May 7, 2026

Type Package

Title Interactive & Customisable Maps using the 'MapLibre GL JS' Library

Version 0.1.0

Description Create interactive maps that can keep up with complex visualisations and large datasets, with this useful interface to the 'MapLibre GL JS' (<https://maplibre.org/maplibre-gl-js/docs/>) library. Users can create maps directly in the console, or as an HTML widget within 'Shiny' web applications, and render spatial data quickly with many customisable options (clusters, custom icons, map layers, and backgrounds). The goal of the package is to make it easier to interpret and explore large spatial datasets within the context of a 'Shiny' dashboard, without having long loading times waiting for a map to update with new data.

URL https://epi-interactive-ltd.github.io/toro/, https://github.com/Epi-interactive-Ltd/toro/

BugReports https://github.com/Epi-interactive-Ltd/toro/issues

License AGPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.5.0)

Imports htmlwidgets, shiny, jsonlite, geojsonsf, base64enc, sf

Suggests knitr, usethis, rmarkdown, lintr, devtools, dplyr, webshot2, mapview, webshot, devtools, spData, testthat (>= 3.0.0)

Config/Needs/website rmarkdown

Config/testthat/edition 3

NeedsCompilation no

Author Poppy Pakinui [aut, cre], Jocelyn Qian [ctb], Nick Snellgrove [ctb], MapLibre contributors [cph] (MapLibre GL JS library), Turf contributors [cph] (turf plugin), Mapbox contributors [cph] (mapbox-gl-draw plugin), epi [cph, fnd]

Maintainer Poppy Pakinui <poppy@epi.group>

Repository https://cran.r-universe.dev

Date/Publication 2026-05-07 19:08:05 UTC

RemoteUrl https://github.com/cran/toro

RemoteRef HEAD

RemoteSha 06b8375beaa1f5ee7fa2128043dfb66e5db51a50

Contents

`add_animation_controls` *Add animation controls to a toro map*

Description

Adds play/pause/stop buttons to control route animations on the map. Optionally includes a speed control slider for adjusting animation speed.

Usage

```
add_animation_controls(
  map,
  route_id = NULL,
  position = "top-right",
  panel_id = NULL,
  buttons = c("play", "pause"),
  include_speed_control = FALSE,
  speed_values = c(0.5, 1, 2),
  speed_labels = c("Slow", "Normal", "Fast"),
  settings = list()
)
```

Arguments

map	A toro map object or a map proxy object.
route_id	Optional route ID to control. If NULL, controls all routes.
position	Position of the controls on the map. Default is "top-right".
panel_id	Optional control panel ID to add controls to instead of map.
buttons	Character vector of buttons to include. Options: "play", "pause", "stop". Default is c("play", "pause").
include_speed_control	Logical. Whether to include a speed control slider. Default is FALSE.
speed_values	Numeric vector of speed values for the speed slider. Default is c(0.5, 1, 2) for slow, normal, and fast speeds.
speed_labels	Character vector of labels for speed values. Default is c("Slow", "Normal", "Fast").
settings	A list of additional settings for the controls.

Value

The map or map proxy object for chaining.

Examples

```
library(sf)

line_data <- sf::st_sf(
  id = 1,
  geometry = sf::st_sfc(
    sf::st_linestring(
      cbind(c(172.2041, 163.9383), c(-32.56960, -46.43999))
    ),
    crs = 4326
  )
)
```

```
map() |>
  add_route(route_id = "route_line", points = line_data) |>
  add_animation_controls(route_id = "route_line", include_speed_control = TRUE)
```

`add_circle_layer` *Add a circle layer to a map or map proxy*

Description

Add a circle layer to a map or map proxy

Usage

```
add_circle_layer(
  map,
  id,
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

`map`	The map object or map proxy to which the layer will be added.
`id`	A unique identifier for the layer.
`source`	The data source for the layer, if not a GeoJSON, it will be converted.
`paint`	A list of paint options for styling the layer. See `get_paint_options()` for defaults and options.
`layout`	A list of layout options for the layer. See `get_layout_options()` for defaults and options.
`popup_column`	The column name to use for popups. Default is `NULL`.
`hover_column`	The column name to use for hover effects. Default is `NULL`.
`can_cluster`	Whether the layer can be clustered. Default is `FALSE`.
`under_id`	The ID of an layer already on the map to place this layer under. Default is `NULL`.
`filter`	A filter expression to apply to the layer. Default is `NULL`. See `get_layer_filter()` for more details on how to create filter expressions.
`...`	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if `can_cluster` is `TRUE`. See the cluster vignette for details on available options.

Value

The updated map object with the circle layer added.

Examples

```
# Load libraries
library(spData)
library(sf)

nz_data <- spData::nz_height |>
  sf::st_transform(4326)

map() |>
 set_bounds(bounds = nz_data) |>
 add_circle_layer(
   id = "nz_elevation",
   source = nz_data,
   hover_column = "elevation"
 )

map() |>
 set_bounds(bounds = nz_data) |>
 add_circle_layer(
   id = "nz_elevation",
   source = nz_data,
   hover_column = "elevation",
   paint = get_paint_options(
     "circle",
     options = list(
       colour = get_column_steps(
         "elevation",
         c(3000),
         c("grey", "black")
       )
     )
   )
 )
```

`add_cluster_toggle` *Add a cluster toggle control to the map or control panel*

Description

Creates a toggle button that can enable/disable clustering for a specific layer.

Usage

```
add_cluster_toggle(
  map,
```

```
  layer_id,
  control_id = NULL,
  left_label = "Toggle Clustering",
  right_label = NULL,
  initial_state = FALSE,
  position = "top-right",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
layer_id	ID of the layer to toggle clustering for.
control_id	ID for the control. If NULL, defaults to "cluster-toggle-<layer_id>".
left_label	Label text for the toggle button. Default is "Toggle Clustering".
right_label	Label text for the toggle button when clustering is off. Default is "Clustering Off".
initial_state	Initial clustering state. Default is FALSE.
position	Position on the map if not using a control panel. Default is "top-right".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

map() |>
 add_symbol_layer(
   id = "quakes",
   source = quakes_data,
   can_cluster = TRUE
 ) |>
 add_cluster_toggle(layer_id = "quakes")
```

add_control_group *Add a control group to a control panel*

Description

Creates a collapsible group within a control panel that can contain multiple controls.

Usage

```
add_control_group(
  map,
  panel_id,
  group_id,
  group_title = NULL,
  collapsible = FALSE,
  collapsed = FALSE
)
```

Arguments

map	The map or map proxy object.
panel_id	ID of the target control panel.
group_id	Unique identifier for the control group.
group_title	Title for the control group (optional).
collapsible	Whether the group can be collapsed. Default is FALSE.
collapsed	Initial collapsed state. Default is FALSE.

Value

The map or map proxy object for chaining.

Examples

```
map() |>
  add_control_panel(panel_id = "my_panel", direction = "row") |>
  add_control_group(
    panel_id = "my_panel",
    group_id = "group_1",
    group_title = "Group 1"
  ) |>
  add_control_group(
    panel_id = "my_panel",
    group_id = "group_2",
    group_title = "Group 2"
  ) |>
  add_cursor_coords_control(panel_id = "my_panel", group_id = "group_1") |>
  add_zoom_control(panel_id = "my_panel", group_id = "group_2")
```

```
map() |>
  add_control_panel(
    panel_id = "my_panel",
    title = "Map Settings",
    position = "top-right",
    collapsible = TRUE,
    collapsed = TRUE,
    direction = "row"
  ) |>
  add_cursor_coords_control(
    panel_id = "my_panel",
    section_title = "Cursor Coordinates"
  )
```

`add_control_panel` *Add a control panel to the map*

Description

Creates a flexible control panel that can contain multiple controls.

Usage

```
add_control_panel(
  map,
  panel_id,
  title = NULL,
  position = "bottom-left",
  collapsible = FALSE,
  collapsed = FALSE,
  direction = "column",
  custom_controls = NULL
)
```

Arguments

`map`	The map or map proxy object.
`panel_id`	Unique identifier for the control panel.
`title`	Title for the control panel. If NULL, no title is shown.
`position`	Position of the control panel on the map. Default is "bottom-left". Options include "top-left", "top-right", "bottom-left", "bottom-right".
`collapsible`	Whether the panel can be collapsed. Default is FALSE.
`collapsed`	Initial collapsed state. Default is FALSE.
`direction`	Layout direction for controls within the panel. Either "row" or "column". Default is "column".
`custom_controls`	List of custom controls to add initially. Each should be a list with elements: html, id (optional), title (optional).

Value

The map or map proxy object for chaining.

Examples

```
map() |>
  add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
  add_cursor_coords_control(panel_id = "my_panel") |>
  add_zoom_control(panel_id = "my_panel")

map() |>
  add_control_panel(
    panel_id = "my_panel",
    title = "Map Settings",
    position = "top-right",
    collapsible = TRUE,
    collapsed = TRUE,
    direction = "row"
  ) |>
  add_cursor_coords_control(
    panel_id = "my_panel",
    section_title = "Cursor Coordinates"
  )
```

---

add_cursor_coords_control

Add a cursor coordinates control to the map

Description

Add a cursor coordinates control to the map

Usage

```
add_cursor_coords_control(
  map,
  position = "bottom-left",
  long_label = "Lng",
  lat_label = "Lat",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.

position	The position of the cursor coordinates control on the map. Default is "bottom-left". Options include "top-left", "top-right", "bottom-left", "bottom-right".
long_label	The label for the longitude coordinate. Default is "Lng".
lat_label	The label for the latitude coordinate. Default is "Lat".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Add to a map
map() |>
 add_cursor_coords_control()

# Change default options
map() |>
 add_cursor_coords_control(
   position = "top-right",
   long_label = "Longitude",
   lat_label = "Latitude"
 )

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_cursor_coords_control(panel_id = "my_panel", section_title = "Cursor Coordinates")

# Add to a control panel inside a control group
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_control_group(
   panel_id = "my_panel",
   group_id = "map_state",
   group_title = "Map State"
   ) |>
 add_cursor_coords_control(panel_id = "my_panel", group_id = "map_state")
```

---

add_custom_control *Add a custom HTML control to the map*

Description

Add a custom HTML control to the map

Usage

```
add_custom_control(
  map,
  id,
  html,
  position = "top-right",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
id	The ID for the custom control.
html	The HTML content to add as a control.
position	The position of the control on the map. Default is "top-right". Options include "top-left", "top-right", "bottom-left", "bottom-right".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Add to a map
map() |>
 add_custom_control(
   id = "custom_control",
   html = "<p>I am a custom control</p>"
)

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_custom_control(
   id = "custom_control_panel",
   html = "<p>I am a custom control in a panel</p>",
   panel_id = "my_panel",
   section_title = "Custom Control Section"
 )

# Add to a control panel inside a control group
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
```

```
add_control_group(
  panel_id = "my_panel",
  group_id = "custom_controls",
  group_title = "Custom Controls"
  ) |>
add_custom_control(
  id = "custom_control_panel",
  html = "<p>I am a custom control in a panel</p>",
  panel_id = "my_panel",
  group_id = "custom_controls"
)
```

`add_draw_control` *Add a draw control to the map*

Description

The draw control allows users to draw shapes (polygons, lines, points) on the map. The drawn shapes can be styled and managed through the control options. Information about the drawn shapes can be retrieved in Shiny using the `input$map_shape_created` (where `map` is the ID of the map) reactive value.

Usage

```
add_draw_control(
  map,
  id = "draw_control",
  position = "top-right",
  modes = c("polygon"),
  active_colour = "#04AAC1",
  inactive_colour = "#04AAC1",
  mode_labels = list(),
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

`map`	The map or map proxy object.
`id`	The ID for the draw control.
`position`	The position of the draw control on the map. Default is `"top-right"`. Options are "top-left", "top-right", "bottom-left", "bottom-right".
`modes`	A vector of modes to enable in the draw control. Default is `c("polygon")`. Options include "polygon", "delete", "line", and "point".
`active_colour`	The colour for the drawn shapes. Default is `"#04AAC1"`.

inactive_colour
: The colour for the inactive shapes. Default is "#04AAC1".

mode_labels
: A named list of labels for each mode. For example, list(polygon = "Draw Polygon", delete = "Delete Shape").

panel_id
: ID of control panel to add to (optional).

section_title
: Section title when added to a control panel.

group_id
: Optional group ID for grouping controls within a panel.

Details

For a more in-depth example see the Draw control article.

Value

The map or map proxy object for chaining.

See Also

get_drawn_shape() to retrieve the drawn shape as an sf object in Shiny.

Examples

```
map() |>
 add_draw_control()
```

---

add_feature_server_source

Add a FeatureService source to the map

Description

Add a FeatureService source to the map

Usage

```
add_feature_server_source(
  map,
  source_url,
  source_id,
  append_query_url = "/0/query?where=1=1&outFields=*&f=geojson"
)
```

Arguments

map
: The map or map proxy object.

source_url
: The URL of the FeatureService source.

source_id
: The ID for the source.

append_query_url
: The query URL to append to the source URL. Default is "/0/query?where=1=1&outFields=*&f=geojs

Value

The map or map proxy object for chaining.

Note

By default the function appends a query URL to the provided `source_url` to retrieve all features in GeoJSON format. If you need more control over the query parameters, you can provide the full query URL directly in the `source_url` argument and set `append_query_url` to an empty string to prevent appending the default query parameters.

Examples

```
service_url <- paste0(
 "https://services1.arcgis.com/VwarAUbcaX64Jhub/arcgis/rest/services/",
 "World_Exclusive_Economic_Zones_Boundaries/FeatureServer"
)

 map() |>
   add_feature_server_source(service_url, "eez") |>
   add_line_layer(id = "eez_lines", source = "eez")
```

---

add_fill_layer *Add a fill layer to a map or map proxy*

Description

Add a fill layer to a map or map proxy

Usage

```
add_fill_layer(
  map,
  id,
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

map	The map object or map proxy to which the layer will be added.
id	A unique identifier for the layer.
source	The data source for the layer, if not a GeoJSON, it will be converted.
paint	A list of paint options for styling the layer. See get_paint_options() for defaults and options.
layout	A list of layout options for the layer. See get_layout_options() for defaults and options.
popup_column	The column name to use for popups. Default is NULL.
hover_column	The column name to use for hover effects. Default is NULL.
can_cluster	Whether the layer can be clustered. Default is FALSE.
under_id	The ID of an layer already on the map to place this layer under. Default is NULL.
filter	A filter expression to apply to the layer. Default is NULL. See get_layer_filter() for more details on how to create filter expressions.
...	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if can_cluster is TRUE. See the cluster vignette for details on available options.

Value

The updated map object with the fill layer added.

Examples

```
# Load libraries
library(dplyr)
library(spData)
library(sf)

nz_data <- spData::nz |>
  dplyr::rename(geometry = geom) |>
  sf::st_transform(4326)

map() |>
 add_fill_layer(
   id = "nz_regions",
   source = nz_data,
   hover_column = "Name"
 )

map() |>
 add_fill_layer(
   id = "nz_regions",
   source = nz_data,
   hover_column = "Name",
   paint = get_paint_options(
     "fill",
```

```
    options = list(
      colour = "#a3b18a",
      opacity = 0.3,
      outline_colour = "#588157"
    )
  )
)
```

---

add_image *Add an image source to the map*

Description

Add an image source to the map

Usage

```
add_image(map, image_id, image_url)
```

Arguments

map	The map or map proxy object.
image_id	The ID of the image source.
image_url	The URL of the image to add.

Value

The map or map proxy object for chaining.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

image_url <- paste0(
 "https://upload.wikimedia.org/wikipedia/en/thumb/0/02/",
 "Leaf_icon.png/600px-Leaf_icon.png"
)

 map() |>
   add_image(
     image_id = "leaf-icon",
     image_url = image_url
   ) |>
   add_symbol_layer(
```

```
    id = "leaf_symbols",
    source = sf::st_as_sf(quakes_data, coords = c("long", "lat"), crs = 4326),
    layout = get_layout_options(
      "symbol",
      options = list(
        icon_image = "leaf-icon",
        icon_size = 0.1
      )
    )
  )
```

`add_lat_lng_grid` *Add a grid of latitude and longitude lines to the map*

Description

Add a grid of latitude and longitude lines to the map

Usage

```
add_lat_lng_grid(map, grid_colour = "#000000")
```

Arguments

`map`	The map or map proxy object.
`grid_colour`	The colour of the grid lines. Default is `"#000000"`.

Value

The map or map proxy object for chaining.

Examples

```
map() |>
 add_lat_lng_grid()
```

`add_layer` *Add a layer to a map or map proxy*

Description

Add a layer to a map or map proxy

Usage

```
add_layer(
  map,
  id,
  type = "fill",
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

`map`	The map object or map proxy to which the layer will be added.
`id`	A unique identifier for the layer.
`type`	The type of layer to add (e.g., "fill", "circle", "line"). Default is "fill".
`source`	The data source for the layer, if not a GeoJSON, it will be converted.
`paint`	A list of paint options for styling the layer. See `get_paint_options()` for defaults and options.
`layout`	A list of layout options for the layer. See `get_layout_options()` for defaults and options.
`popup_column`	The column name to use for popups. Default is `NULL`.
`hover_column`	The column name to use for hover effects. Default is `NULL`.
`can_cluster`	Whether the layer can be clustered. Default is `FALSE`.
`under_id`	The ID of an layer already on the map to place this layer under. Default is `NULL`.
`filter`	A filter expression to apply to the layer. Default is `NULL`. See `get_layer_filter()` for more details on how to create filter expressions.
`...`	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if `can_cluster` is `TRUE`. See the cluster vignette for details on available options.

Value

The updated map object with the new layer added.

Note

If `source` is not a string referring to an existing source, it will be converted to a GeoJSON source and added to the map automatically. The Id for the source is generated by appending the layer Id to `source-`. For example, if you add a layer with `id = "my_layer"` and `source` is a sf object, the source is added to the map with the ID `"source-my_layer"`.

See Also

`get_paint_options()` for paint customisation, `get_layout_options()` for layout customisation, and `get_layer_filter()` for applying filters to layers.

Examples

```
# Load libraries
library(spData)
library(dplyr)
library(sf)

nz_data <- spData::nz |>
  dplyr::rename(geometry = geom) |>
  sf::st_transform(4326)

map() |>
 add_layer(
   id = "nz_regions",
   type = "fill",
   source = nz_data,
   hover_column = "Name"
 )
```

---

add_layer_selector_control

Add a layer selector control to the map or control panel

Description

Creates a drop-down selector that allows switching between layers, showing only the selected layer while hiding all others. This is useful for comparing different data layers or allowing users to choose between mutually exclusive visualizations.

Usage

```
add_layer_selector_control(
  map,
  layer_ids,
  labels = NULL,
  default_layer = NULL,
  none_option = FALSE,
  none_label = "None",
  position = "top-right",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
layer_ids	Vector of layer IDs to include in the selector.
labels	Named vector of labels for layers. If NULL, uses layer IDs directly.
default_layer	Default layer to select. If NULL, uses the first layer.
none_option	Whether to include a "None" option that hides all layers. Default is FALSE.
none_label	Label for the "None" option. Default is "None".
position	Position on the map if not using a control panel. Default is "top-right".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Load libraries
library(spData)
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

nz_data <- spData::nz_height |>
  sf::st_transform(4326)

map() |>
 add_circle_layer(
```

```
  id = "quakes",
  source = quakes_data
) |>
add_circle_layer(
  id = "nz_elevation",
  source = nz_data
) |>
add_layer_selector_control(
  layer_ids = c("quakes", "nz_elevation"),
  labels = c("quakes" = "Earthquakes", "nz_elevation" = "NZ Elevation")
)
```

`add_line_layer` *Add a line layer to a map or map proxy*

Description

Add a line layer to a map or map proxy

Usage

```
add_line_layer(
  map,
  id,
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

`map`	The map object or map proxy to which the layer will be added.
`id`	A unique identifier for the layer.
`source`	The data source for the layer, if not a GeoJSON, it will be converted.
`paint`	A list of paint options for styling the layer. See `get_paint_options()` for defaults and options.
`layout`	A list of layout options for the layer. See `get_layout_options()` for defaults and options.
`popup_column`	The column name to use for popups. Default is `NULL`.
`hover_column`	The column name to use for hover effects. Default is `NULL`.

can_cluster	Whether the layer can be clustered. Default is FALSE.
under_id	The ID of an layer already on the map to place this layer under. Default is NULL.
filter	A filter expression to apply to the layer. Default is NULL. See get_layer_filter() for more details on how to create filter expressions.
...	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if can_cluster is TRUE. See the cluster vignette for details on available options.

Value

The updated map object with the line layer added.

Examples

```
# Load libraries
library(spData)
library(sf)

seine_data <- spData::seine |>
  sf::st_transform(4326)

map() |>
 set_bounds(bounds = seine_data, padding = 100) |>
 add_line_layer(
   id = "seine_lines",
   source = seine_data,
   hover_column = "name"
 )

map() |>
 set_bounds(bounds = seine_data, padding = 100) |>
 add_line_layer(
   id = "seine_lines",
   source = seine_data,
   hover_column = "name",
   paint = get_paint_options(
     "line",
     options = list(
       colour = get_column_group(
         "name",
         c("Marne" = "#014f86", "Seine" = "#61a5c2"),
         "#a9d6e5"
       ),
       line_width = get_column_group("name", c("Seine" = 3), 1)
     )
   )
 )
```

add_route *Add a route to a toro map which can be animated*

Description

A route is a line that can be animated along a set of points. This function adds a route to the map with a unique identifier and settings for the route's appearance and animation.

Usage

```
add_route(map, route_id, points, settings = list())
```

Arguments

map	A toro map object or a map proxy object.
route_id	A unique identifier for the route.
points	A sf object containing the points of the route.
settings	A list of settings for the route (e.g., color, weight).

Value

The map or map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)
library(sf)

line_data <- sf::st_sf(
  id = 1,
  geometry = sf::st_sfc(
    sf::st_linestring(
      cbind(c(172.2041, 163.9383), c(-32.56960, -46.43999))
    ),
    crs = 4326
  )
)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("play_route", "Play Route Animation"),
   actionButton("pause_route", "Pause Route Animation"),
   actionButton("remove_route", "Remove Route")
 )
)
```

```
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_route(route_id = "route_line", points = line_data)
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     play_route(route_id = "route_line")
 }) |>
   bindEvent(input$play_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     pause_route(route_id = "route_line")
 }) |>
   bindEvent(input$pause_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_route(route_id = "route_line")
 }) |>
   bindEvent(input$remove_route)
}
}
```

`add_source` *Add a source to the map*

Description

Add a source to the map

Usage

```
add_source(map, source_id, data, type = "geojson", cluster = FALSE, ...)
```

Arguments

`map`	The map or map proxy object.
`source_id`	The ID for the source.
`data`	The data for the source, typically in GeoJSON format.
`type`	The type of the source. Default is `"geojson"`. Other options include `"vector"` or `"raster"`.
`cluster`	Whether to enable clustering for this source. Default is `FALSE`.

... Additional arguments to in pass directly to the JS addSource function. Documentation for this can be found on the MapLibre GL JS docs.

- id: Alternative to `source_id` for backward compatibility. If both `source_id` and `id` are provided, `source_id` will take precedence.

Value

The map or map proxy object for chaining.

Note

If you add a source directly in an add layer function, the source ID will be automatically generated as `source-{layer-id}`.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

# Display the source on map
map() |>
  add_source(
    source_id = "my_source",
    data = sf::st_as_sf(quakes_data, coords = c("long", "lat"), crs = 4326)
  ) |>
  add_circle_layer(id = "quakes", source = "my_source")
```

---

add_speed_control *Add a speed control to the map or control panel*

Description

The speed control allows users to adjust the speed of an animation on the map, such as a time-based animation controlled by the timeline control. It can be added as a standalone control on the map or within a control panel for better organization of multiple controls.

Usage

```
add_speed_control(
  map,
  values = c(0.5, 1, 2),
  labels = c("Slow", "Normal", "Fast"),
  default_index = 2,
  position = "top-right",
```

```
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
values	Vector of speed multiplier values. Default is c(0.5, 1, 2).
labels	Vector of labels for each speed value. Default is c("Slow", "Normal", "Fast").
default_index	Index of the default speed (1-based). Default is 2.
position	Position on the map if not using a control panel. Default is "top-right".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	Optional ID of the group to add the control to within a panel.

Value

The map or map proxy object for chaining.

Examples

```
# Add to a map (no dates specified)
map() |>
 add_speed_control()

# Change default options
map() |>
 add_speed_control(
   values = c(0.5, 1, 2, 5),
   labels = c("Slow", "Normal", "Fast", "Super fast"),
   default_index = 4 # Start on Super fast
 )

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_speed_control(panel_id = "my_panel")

# Add to a control panel inside a control group
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_control_group(
   panel_id = "my_panel",
   group_id = "animation_controls",
   group_title = "Animation Controls"
 ) |>
 add_speed_control(
   panel_id = "my_panel",
```

```
  group_id = "animation_controls"
)
```

`add_symbol_layer` *Add a symbol layer to a map or map proxy*

Description

This layer is typically used for icons or pins.

Usage

```
add_symbol_layer(
  map,
  id,
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

`map`	The map object or map proxy to which the layer will be added.
`id`	A unique identifier for the layer.
`source`	The data source for the layer, if not a GeoJSON, it will be converted.
`paint`	A list of paint options for styling the layer. See `get_paint_options()` for defaults and options.
`layout`	A list of layout options for the layer. See `get_layout_options()` for defaults and options.
`popup_column`	The column name to use for popups. Default is `NULL`.
`hover_column`	The column name to use for hover effects. Default is `NULL`.
`can_cluster`	Whether the layer can be clustered. Default is `FALSE`.
`under_id`	The ID of an layer already on the map to place this layer under. Default is `NULL`.
`filter`	A filter expression to apply to the layer. Default is `NULL`. See `get_layer_filter()` for more details on how to create filter expressions.
`...`	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if `can_cluster` is `TRUE`. See the cluster vignette for details on available options.

Value

The updated map object with the symbol layer added.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

# Create map and add fill layer
map() |>
 add_symbol_layer(
   id = "test_layer",
   source = quakes_data
 )
```

---

add_text_layer *Add a text layer to a map or map proxy*

Description

This layer is typically used for text labels.

Usage

```
add_text_layer(
  map,
  id,
  source,
  paint = NULL,
  layout = NULL,
  popup_column = NULL,
  hover_column = NULL,
  can_cluster = FALSE,
  under_id = NULL,
  filter = NULL,
  ...
)
```

Arguments

map	The map object or map proxy to which the layer will be added.
id	A unique identifier for the layer.

source	The data source for the layer, if not a GeoJSON, it will be converted.
paint	A list of paint options for styling the layer. See get_paint_options() for defaults and options.
layout	A list of layout options for the layer. See get_layout_options() for defaults and options.
popup_column	The column name to use for popups. Default is NULL.
hover_column	The column name to use for hover effects. Default is NULL.
can_cluster	Whether the layer can be clustered. Default is FALSE.
under_id	The ID of an layer already on the map to place this layer under. Default is NULL.
filter	A filter expression to apply to the layer. Default is NULL. See get_layer_filter() for more details on how to create filter expressions.
...	Additional arguments to include in the layer definition. • clusterOptions: A list of options for clustering, if can_cluster is TRUE. See the cluster vignette for details on available options.

Value

The updated map object with the text layer added.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

# Create map and add fill layer
map() |>
 add_text_layer(
   id = "test_layer",
   source = quakes_data
 )
```

---

add_tile_selector_control

Add a tile selector control to the map or control panel

Description

Add a tile selector control to the map or control panel

Usage

```
add_tile_selector_control(
  map,
  available_tiles = NULL,
  labels = NULL,
  default_tile = NULL,
  position = "top-right",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
available_tiles	Vector of available tile options. If NULL, uses all loaded tiles from the map.
labels	Named vector of labels for tiles. If NULL, uses tile names directly.
default_tile	Default tile to select. If NULL, uses current map tile.
position	Position on the map if not using a control panel. Default is "top-right".
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Add a tile selector that gives the ability to switch between all available tilesets
all_tiles <- get_tile_options()

map(loadedTiles = all_tiles) |>
 add_tile_selector_control(available_tiles = all_tiles)
```

---

add_timeline_control *Add a timeline control to the map or control panel*

Description

The timeline control allows for users to interact with a date-based timeline, and can be used to control an animation of map data over time.

Usage

```
add_timeline_control(
  map,
  start_date = NULL,
  end_date = NULL,
  position = "bottom-left",
  max_ticks = 3,
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
start_date	Start date for the timeline (YYYY-MM-DD format).
end_date	End date for the timeline (YYYY-MM-DD format).
position	Position on the map if not using a control panel. Default is "bottom-left".
max_ticks	Maximum number of labeled ticks to prevent overlap. Default is 3.
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	Optional ID of the group to add the control to within a panel.

Value

The map or map proxy object for chaining.

Examples

```
# Add to a map (no dates specified)
map() |>
 add_timeline_control()

# Add to map with dates
map() |>
 add_timeline_control(
   start_date = Sys.Date(),
   end_date = Sys.Date() + 30
 )

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_timeline_control(
   start_date = Sys.Date(),
   end_date = Sys.Date() + 30,
   panel_id = "my_panel"
 )
```

```
# Add to a control panel inside a control group
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_control_group(
   panel_id = "my_panel",
   group_id = "animation_controls",
   group_title = "Animation Controls"
 ) |>
 add_timeline_control(
   start_date = Sys.Date(),
   end_date = Sys.Date() + 30,
   panel_id = "my_panel",
   group_id = "animation_controls"
 )
```

`add_visibility_toggle` *Add a visibility toggle control to the map or control panel*

Description

Creates a toggle button that can show/hide a specific layer.

Usage

```
add_visibility_toggle(
  map,
  layer_id,
  control_id = NULL,
  left_label = "Toggle Layer",
  right_label = NULL,
  initial_state = TRUE,
  position = "top-right",
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

`map`	The map or map proxy object.
`layer_id`	ID of the layer to toggle visibility for.
`control_id`	ID for the control. If `NULL`, defaults to "visibility-toggle-<layer_id>".
`left_label`	Label text for the toggle button. Default is "Toggle Layer".
`right_label`	Label text for the toggle button when layer is hidden. Default is "Layer Hidden".
`initial_state`	Initial visibility state. Default is `TRUE`.
`position`	Position on the map if not using a control panel. Default is "top-right".

panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	ID of control group to add to (optional).

Value

The map or map proxy object for chaining.

Examples

```
# Load libraries
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

map() |>
 add_circle_layer(
   id = "quakes",
   source = quakes_data
 ) |>
 add_visibility_toggle(layer_id = "quakes")
```

---

add_zoom_control *Add a zoom control to the map*

Description

Add a zoom control to the map

Usage

```
add_zoom_control(
  map,
  position = "top-right",
  control_options = list(),
  panel_id = NULL,
  section_title = NULL,
  group_id = NULL
)
```

Arguments

map	The map or map proxy object.
position	The position of the zoom control on the map. Default is "top-right".
control_options	Additional options for the zoom control. Default is an empty list.
panel_id	ID of control panel to add to (optional).
section_title	Section title when added to a control panel.
group_id	Optional ID of the group to add the control to within a panel.

Value

The map proxy object for chaining.

Note

See MapLibre NavigationControl docs for more information on available options.

Examples

```
add_zoom_control(map())

# Inside a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "View Controls") |>
 add_zoom_control(panel_id = "my_panel")
```

---

delete_drawn_shape *Delete a drawn shape from the map*

Description

The ID of the shape is provided by the draw control when a shape is created.

Usage

```
delete_drawn_shape(proxy, shape_id)
```

Arguments

proxy	The map proxy object created by mapProxy().
shape_id	The ID of the shape to delete.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   selectInput("shape_ids", "Drawn shape IDs", choices = NULL),
   actionButton("remove_drawn_shape", "Remove drawn shape")
 )
)
server <- function(input, output, session) {
 drawn_shape_ids <- reactiveVal(character())

 output$map <- renderMap({
   map() |>
    add_draw_control()
 })

 # Update the select input options with current shape IDs
 observe({
   req(input$map_loaded)
   updateSelectInput(inputId = "shape_ids", choices = drawn_shape_ids())
 })

 # Update the list of drawn shape IDs when a new shape is created
 observe({
   req(input$map_loaded, input$map_shape_created)
   new_shape <- get_drawn_shape(input$map_shape_created)
   drawn_shape_ids(c(drawn_shape_ids(), new_shape$id))
 }) |>
   bindEvent(input$map_shape_created)

 # Delete the selected drawn shape when the button is clicked
 observe({
   req(input$map_loaded, input$shape_ids)
   mapProxy("map") |>
     delete_drawn_shape(input$shape_ids)
   # Remove the deleted shape ID from the list of drawn shape IDs
   drawn_shape_ids(setdiff(drawn_shape_ids(), input$shape_ids))
 }) |>
   bindEvent(input$remove_drawn_shape)
}
}
```

`export_map_image` *Export map as an image (non-Shiny context)*

Description

This function exports a map widget as an image file using webshot2 or mapview. Works in non-Shiny contexts like RMarkdown, scripts, or interactive sessions.

Usage

```
export_map_image(
  map,
  filepath,
  width = 800,
  height = 600,
  delay = 2,
  zoom = 1,
  ...
)
```

Arguments

map	A map object created by map().
filepath	The file path to save the image (including extension).
width	The width of the image in pixels. Default is 800.
height	The height of the image in pixels. Default is 600.
delay	The delay in seconds before capturing. Default is 2.
zoom	The zoom factor for the capture. Default is 1.
...	Additional arguments passed to webshot2::webshot() or mapview::mapshot().

Value

The file path of the saved image (invisibly).

Examples

```
# Load library
library(sf)

data <- data.frame(lon = 174.8210, lat = -41.3096) |>
  sf::st_as_sf(coords = c("lon", "lat"), crs = 4326)
# Create and export a map
my_map <- map() |>
  add_circle_layer("epi_circle", source = data)

export_map_image(my_map, file.path(tempdir(), "my_map.png"), width = 1200, height = 800)
```

`get_clicked_feature` *Get the sf data frame of a clicked feature from the map widget*

Description

The click input is a list containing the `layerId`, `properties`, `geometry`, and `time`. Turn this into an `sf` object.

Usage

```
get_clicked_feature(clicked_feature_input)
```

Arguments

`clicked_feature_input`
: A list representing the clicked feature.

Details

To get the clicked feature, use `input$map_feature_click` in a Shiny app, where `map` is the ID of your map output. This input will contain a list representing the clicked feature whenever a feature is clicked on the map. Pass this input to `get_clicked_feature()` to convert it into an `sf` object for easier manipulation in R.

Value

A `sf` object representing the clicked feature, or `NULL`.

Note

`time` is not used in this function, but it is included in the input so that the same feature can be clicked multiple times and the changed time means that the input will be updated.

Examples

```
if(interactive()){
library(shiny)
library(spData)
library(sf)
library(toro)

nz_data <- spData::nz_height |>
  sf::st_transform(4326)

ui <- fluidPage(
 tagList(
   mapOutput("map")
 )
)
```

```
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     set_bounds(bounds = nz_data) |>
     add_circle_layer(
       id = "nz_elevation",
       source = nz_data
     )
 })

 # Print the clicked feature as an sf object
 observe({
   req(input$map_loaded, input$map_feature_click)
   print(get_clicked_feature(input$map_feature_click))
 }) |>
   bindEvent(input$map_feature_click)
}
}
```

`get_column` *Get a column from a dataset to use as a paint or layout option in a map layer*

Description

Allows the column value to be used for styling features in a map layer.

Usage

```
get_column(column_name)
```

Arguments

`column_name`	String representing the name of the column to be used.

Value

List containing the paint or layout option to be set.

Examples

```
get_column("opacity")
get_column("icon")
# Use in a paint property: list("circle-color" = get_column("color"))
```

get_column_boolean *Get the value for a paint or layout option in a map layer based on a column boolean value*

Description

Allows the data to be styled by the group option in the column.

Usage

```
get_column_boolean(column_name, true_value, false_value)
```

Arguments

column_name	String representing the name of the column to be used.
true_value	Value to use when the column value is TRUE.
false_value	Value to use when the column value is FALSE.

Value

List containing the paint or layout option to be set.

Examples

```
get_column_boolean("group", "red", "grey")
```

get_column_group *Get the values for a paint or layout option in a map layer based on a column value*

Description

Allows the data to be styled by the group option in the column.

Usage

```
get_column_group(column_name, named_group_values, default_value = "#cccccc")
```

Arguments

column_name	String representing the name of the column to be used.
named_group_values	Vector of value strings named by the group values. The names of the vector should match the group values in the column.
default_value	String for the default value to use if no match is found. Default is "#cccccc".

Value

List containing the paint or layout option to be set.

Note

If using numbers as the group values, then you need to use `stats::setNames` rather than a named vector, as the names of the vector will be coerced to strings.

Examples

```
get_column_group("group", c("A" = "red", "B" = "blue"), "grey")
get_column_group("opacity", stats::setNames(c(0.3, 0.5), c("A", "B")), 0.6)
```

---

get_column_steps *Get the properties for a column in a map layer based on step breaks*

Description

Allows the data to be styled by the step breaks in the column.

Usage

```
get_column_steps(column_name, breaks, values)
```

Arguments

`column_name`	String representing the name of the column to be used.
`breaks`	Numeric vector of thresholds (must be sorted ascending).
`values`	Vector of values, length = length(breaks) + 1.

Value

List containing the paint or layout option to be set.

Examples

```
get_column_steps("value", c(10, 20, 30), c("red", "orange", "yellow", "green"))
```

get_drawn_shape *Get the drawn shape from the map widget*

Description

Parses the JSON string returned by the map widget when a shape is drawn. Ensures that the ID of the shape is included in the resulting sf object.

Usage

```
get_drawn_shape(create_input_string)
```

Arguments

create_input_string	A JSON string representing the drawn shape.

Details

To get the drawn shape, use input$map_shape_created in a Shiny app, where map is the ID of your map output. This input will contain a JSON string representing the drawn shape whenever a new shape is created using the draw control on the map. Pass this input to get_drawn_shape() to convert it into an sf object for easier manipulation in R.

Value

A sf object representing the drawn shape, or NULL.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
    add_draw_control()
 })

 # Update the list of drawn shape IDs when a new shape is created
 observe({
   req(input$map_loaded, input$map_shape_created)
   new_shape <- get_drawn_shape(input$map_shape_created)
```

```
    print(new_shape)
  }) |>
    bindEvent(input$map_shape_created)
}
}
```

`get_layer_filter` *Get a filter for a layer*

Description

Parse a filter string into a list of filters that the map can use.

Usage

```
get_layer_filter(filter_str)
```

Arguments

`filter_str`	A string or vector of strings representing the filter conditions.

Value

A list where the first element is "all" if multiple filters are provided, or a single filter condition.

Examples

```
# Filter to only show rows where the "layer_id" column is equal to "forests"
get_layer_filter("layer_id == forests")

# Filter to show rows where the "layer_id" column is equal to "sites" and the "project_status"
# column is equal to "Confirmed"
get_layer_filter(c("layer_id == sites", "project_status == Confirmed"))
```

`get_layout_options` *Get layout options for a specific layer type*

Description

This function returns a list of layout options based on the layer type and any additional options provided.

Usage

```
get_layout_options(layer_type, options = list())
```

Arguments

`layer_type` A string indicating the type of layer (e.g., "fill", "circle", "line").

`options` A list of additional options to customize the layout properties. See MapLibre docs for options.

- line_cap: A list of options for the line cap, if `layer_type` is "line". Default is "round".
- line_join: A list of options for the line join, if `layer_type` is "line". Default is "round".
- icon_image: The name of the icon image to use for symbol layers, if `layer_type` is "symbol". Default is "toro-pin".
- icon_size: The size of the icon for symbol layers, if `layer_type` is "symbol". Default is 1.
- icon_anchor: The anchor point for the icon in symbol layers, if `layer_type` is "symbol". Default is "bottom".
- text_anchor: The anchor point for the text in symbol layers, if `layer_type` is "symbol". Default is "center".
- icon_offset: The offset for the icon in symbol layers, if `layer_type` is "symbol". Default is list(0, 0).
- icon_allow_overlap: Whether to allow icons to overlap in symbol layers, if `layer_type` is "symbol". Default is `TRUE`.
- text_allow_overlap: Whether to allow text to overlap in symbol layers, if `layer_type` is "symbol". Default is `TRUE`
- icon_rotate: The rotation angle for icons in symbol layers, if `layer_type` is "symbol". Default is 0.
- icon_flip_horizontal: Whether to flip icons horizontally in symbol layers, if `layer_type` is "symbol". Default is FALSE. Note that this uses the `icon-flip-horizontal` property in MapLibre, which may not be supported by all icons.
- text_font: The font for text in symbol layers, if `layer_type` is "symbol". Default is "Open Sans Regular".
- text_field: The text field for symbol layers, if `layer_type` is "symbol". Default is `NULL`, which means no text will be shown. This can be set to a column value using `get_column()` or other functions to show text from the data.
- text_size: The size of the text in symbol layers, if `layer_type` is "symbol". Default is 12. You can also provide any other layout options found in the MapLibre docs for the specific layer type, and they will be included in the returned list.

Value

A list of layout options suitable for the specified layer type.

Note

You can provide any layout options found in the MapLibre Layers docs in the `options` argument, and they will be included in the returned list. The default options are just a starting point and can be overridden by providing them in the `options` argument.

See Also

`get_column()` for getting options from column values directly, `get_column_group()` for getting options by spiliting column values into groups, and `get_column_steps()` for getting options by spiliting column values into groups based on steps.

Examples

```
get_layout_options("line", list(line_cap = "butt", line_join = "bevel"))

get_layout_options("symbol", list(icon_image = "yellow_pin", icon_size = 1.5))

# For horizontal flipping, provide left/right versions of your icon or use rotation fallback
get_layout_options("symbol", list(icon_image = "arrow", icon_flip_horizontal = TRUE))

# Provide options outside of the defaults
get_layout_options(
 "circle",
 list(
   "circle-sort-key" = get_column_steps(
     "elevation",
     c(3000),
     c(100, 200)
   )
 )
)
```

---

get_paint_options *Get paint options for a specific layer type*

Description

This function returns a list of paint options based on the layer type and any additional options provided.

Usage

```
get_paint_options(layer_type, options = list())
```

Arguments

`layer_type`	A string indicating the type of layer (e.g., "fill", "circle", "line").
`options`	A list of additional options to customize the paint properties. See MapLibre docs for full options. • colour: The color to use for the layer. Default is "grey". `color` is also accepted as an alias for `colour`. • opacity: The opacity to use for the layer. Default is 1 (fully opaque). • line_width: The width of lines in line layers or the outline width for circle layers. Default is 1. • line_dash: The dash pattern for line layers. Default is no dash (solid line). You can provide a vector of numbers to specify the dash pattern (e.g., c(2, 4) for a pattern of 2 units on, 4 units off). • outline_colour: The color to use for the outline of circle or fill layers. Default is the same as `colour`. `outline_color` is also accepted as an alias for `outline_colour`. • outline_opacity: The opacity to use for the outline of circle or fill layers. Default is the same as `opacity`. You can also provide any other paint options found in the MapLibre docs for the specific layer type, and they will be included in the returned list.

Value

A list of paint options suitable for the specified layer type.

Note

You can provide any paint options found in the MapLibre Layers docs in the `options` argument, and they will be included in the returned list. The default options are just a starting point and can be overridden by providing them in the `options` argument.

See Also

`get_column()` for getting options from column values directly, `get_column_group()` for getting options by spiliting column values into groups, and `get_column_steps()` for getting options by spiliting column values into groups based on steps.

Examples

```
get_paint_options("line", list(colour = "blue", opacity = 0.8, line_width = 2))

get_paint_options("circle", list(colour = "red", circle_radius = 10, outline_colour = "black"))

# Use with get_column for data-driven styling:
get_paint_options("fill", list(colour = get_column("color"), opacity = get_column("opacity")))

get_paint_options("fill", list(
   colour = get_column_group("group", c("A" = "green", "B" = "blue"))
))
```

```
get_paint_options("fill", list(
   opacity = get_column_steps("percent", c(25, 75), c("red", "orange", "yellow"))
))

# Provide options outside of the defaults
get_paint_options("circle", list("circle-blur" = 0.5))
```

---

get_tile_options *Get available tile layer options*

Description

Get available tile layer options

Usage

```
get_tile_options()
```

Value

A character vector of available tile layer options.

Examples

```
all_tiles <- get_tile_options()
```

---

hide_layer *Hide a layer from the map*

Description

Hide a layer from the map

Usage

```
hide_layer(proxy, layer_id)
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
layer_id	The ID of the layer to hide.

Value

The map proxy object for chaining.

Note

This does not remove the layer, it only hides it from view.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("show_layer", "Show Layer"),
   actionButton("hide_layer", "Hide Layer")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_circle_layer(
       id = "quakes",
       source = quakes_data
     )
 })

 observe({
   mapProxy("map") |>
     show_layer(layer_id = "quakes")
 }) |>
   bindEvent(input$show_layer)

 observe({
   mapProxy("map") |>
     hide_layer(layer_id = "quakes")
 }) |>
   bindEvent(input$hide_layer)
}
}
```

---

map *Create a MapLibre map widget*

Description

This function creates a map htmlwidget for use in R and Shiny applications.

Usage

```
map(
  style = "lightgrey",
  center = c(174, -41),
  zoom = 2,
  width = "100%",
  height = NULL,
  session = shiny::getDefaultReactiveDomain(),
  ...
)
```

Arguments

style	The style of the map. Default is "lightgrey".
center	The initial center of the map as a longitude/latitude pair. Default is c(174, -41).
zoom	The initial zoom level of the map. Default is 2.
width	The width of the widget. Optional.
height	The height of the widget. Optional.
session	The Shiny session object. Default is the current session.
...	Additional options to customize the map.

- minZoom: Minimum zoom level (0-24). Default is 2.
- maxZoom: Maximum zoom level (0-24). Default is 18.
- clusterColour: The colour of the cluster circles. Default is "#808080".
- loadedTiles: A character vector of tile ids to load, or a named list of tile options. Full options: c("natgeo", "satellite", "topo", "terrain", "streets", "shaded", "lightgrey"). Default is c("lightgrey", "satellite").
- initialTileLayer: The tile layer to use when the map is first loaded. Default is NULL (the first layer in loadedTiles).
- backgroundColour: The background colour of the map. Default is "#D0CFD4".
- enable3D: Whether to enable 3D dragging/view. Default is FALSE.
- initialLoadedLayers: A character vector of layer ids that should be loaded before the initial map spinner is hidden. Default is NULL.
- spinnerWhileBusy: Whether to show a spinner while the map is busy (e.g. loading tiles). Default is FALSE.
- busyLoaderBgColour: The background colour of the busy loader. Default is "rgba(0, 0, 0, 0.2)".
- busyLoaderColour: The colour of the busy loader. Default is "white".
- initialLoaderBgColour: The background colour of the initial loader. Default is "white".
- initialLoaderColour: The colour of the initial loader. Default is "black".
- clusterOptions: A list of options for clustering, if can_cluster is TRUE. See the cluster vignette for details on available options.
- attributionPosition: The position of the attribution control. Default is "bottomright".

Value

An object of class `htmlwidget` representing the map.

Examples

```
map()

# Load two tilesets for the map to use
map(loadedTiles = c("natgeo", "streets"))

# Load two tilesets for the map to use and add maxzoom to satellite layer
map(loadedTiles = list(natgeo = list(), satellite = list(maxZoom = 2)))
```

`mapOutput` *Create a MapLibre GL output for use in Shiny*

Description

Create a MapLibre GL output for use in Shiny

Usage

```
mapOutput(outputId, width = "100%", height = "600px")
```

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.

Value

A MapLibre GL map for use in a Shiny UI.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map()
 })
```

```
}
}
```

---

mapProxy *Create a proxy object for updating the map*

Description

Create a proxy object for updating the map

Usage

```
mapProxy(outputId, session = shiny::getDefaultReactiveDomain())
```

Arguments

outputId	The ID of the output element.
session	The Shiny session object (default is the current session).

Value

A proxy object for the map.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   checkboxInput("has_zoom_controls", "Remove Zoom Controls", value = TRUE)
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_zoom_control()
 })

 observe({
   req(input$map_loaded)
   if (input$has_zoom_controls == TRUE) {
     mapProxy("map") |>
       add_zoom_control()
   } else {
     mapProxy("map") |>
       remove_zoom_control()
```

```
    }
  }) |>
    bindEvent(input$has_zoom_controls)
}
}
```

---

pause_route *Pause a route animation on a toro map*

Description

Pause a route animation on a toro map

Usage

```
pause_route(map, route_id, settings = list())
```

Arguments

map	A toro map proxy object.
route_id	A unique identifier for the route.
settings	A list of settings for pausing the animation.

Value

The updated map proxy object.

Examples

```
if(interactive()){
library(shiny)
library(toro)
library(sf)

line_data <- sf::st_sf(
  id = 1,
  geometry = sf::st_sfc(
    sf::st_linestring(
      cbind(c(172.2041, 163.9383), c(-32.56960, -46.43999))
    ),
    crs = 4326
  )
)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("play_route", "Play Route Animation"),
   actionButton("pause_route", "Pause Route Animation"),
```

```
    actionButton("remove_route", "Remove Route")
  )
)
server <- function(input, output, session) {
  output$map <- renderMap({
    map() |>
      add_route(route_id = "route_line", points = line_data)
  })

  observe({
    req(input$map_loaded)
    mapProxy("map") |>
      play_route(route_id = "route_line")
  }) |>
    bindEvent(input$play_route)

   observe({
    req(input$map_loaded)
    mapProxy("map") |>
      pause_route(route_id = "route_line")
  }) |>
    bindEvent(input$pause_route)

   observe({
    req(input$map_loaded)
    mapProxy("map") |>
      remove_route(route_id = "route_line")
  }) |>
    bindEvent(input$remove_route)
}
}
```

`play_route` *Play a route animation on a toro map*

Description

Play a route animation on a toro map

Usage

```
play_route(map, route_id, settings = list())
```

Arguments

`map`	A toro map proxy object.
`route_id`	A unique identifier for the route.
`settings`	A list of settings for the animation (e.g., speed, loop).

Value

The updated map proxy object.

Examples

```
if(interactive()){
library(shiny)
library(toro)
library(sf)

line_data <- sf::st_sf(
  id = 1,
  geometry = sf::st_sfc(
    sf::st_linestring(
      cbind(c(172.2041, 163.9383), c(-32.56960, -46.43999))
    ),
    crs = 4326
  )
)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("play_route", "Play Route Animation"),
   actionButton("pause_route", "Pause Route Animation"),
   actionButton("remove_route", "Remove Route")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_route(route_id = "route_line", points = line_data)
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     play_route(route_id = "route_line")
 }) |>
   bindEvent(input$play_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     pause_route(route_id = "route_line")
 }) |>
   bindEvent(input$pause_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_route(route_id = "route_line")
```

```
 }) |>
   bindEvent(input$remove_route)
}
}
```

`remove_cluster_toggle` *Remove a cluster toggle control from the map*

Description

Remove a cluster toggle control from the map

Usage

```
remove_cluster_toggle(proxy, layer_id, panel_id = NULL)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`layer_id`	The ID of the layer whose cluster toggle control to remove.
`panel_id`	Optional. If provided, removes the control from the specified control panel.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
# Load libraries
library(shiny)
library(toro)
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_control", "Remove cluster control")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   add_symbol_layer(
     id = "quakes",
```

```
    source = quakes_data,
    can_cluster = TRUE
  ) |>
  add_cluster_toggle(layer_id = "quakes")
})

observe({
  req(input$map_loaded)
  mapProxy("map") |>
    remove_cluster_toggle("quakes")
}) |>
  bindEvent(input$remove_control)
}
}
```

`remove_control` *Remove a control from the map*

Description

Remove a control from the map

Usage

```
remove_control(proxy, control_id)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`control_id`	The ID of the control to remove.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_controls", "Remove controls")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
```

```
  map() |>
    add_zoom_control() |>
    add_custom_control(
      id = "custom_control",
      html = "<p>I am a custom control</p>"
    )
})

observe({
  req(input$map_loaded)
  mapProxy("map") |>
    remove_control("zoom_control") |>
    remove_control("custom_control")
}) |>
  bindEvent(input$remove_controls)
}
}
```

`remove_control_group` *Remove a control group from a control panel*

Description

Remove a control group from a control panel

Usage

```
remove_control_group(proxy, panel_id, group_id)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`panel_id`	The ID of the control panel.
`group_id`	The ID of the control group to remove.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_group1", "Remove control group 1")
```

```
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
     add_control_group(
       panel_id = "my_panel",
       group_id = "control_group1",
       group_title = "Control Group 1"
     ) |>
    add_control_group(
       panel_id = "my_panel",
       group_id = "control_group2",
       group_title = "Control Group 2"
     )
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_control_group(panel_id = "my_panel", group_id = "control_group1")
 }) |>
   bindEvent(input$remove_group1)
}
}
```

`remove_cursor_coords_control` *Remove the cursor coordinates control from the map*

Description

Remove the cursor coordinates control from the map

Usage

```
remove_cursor_coords_control(proxy, panel_id = NULL)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`panel_id`	Optional. If provided, removes the cursor coordinates control from the specified control panel. If `NULL`, removes the standalone cursor coordinates control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   fluidRow(
     column(6, mapOutput("map_one")),
     column(6, mapOutput("map_two"))
   ),
   actionButton("remove_control", "Remove control")
 )
)
server <- function(input, output, session) {
 output$map_one <- renderMap({
   map() |>
     add_cursor_coords_control()
 })

 output$map_two <- renderMap({
   map() |>
     add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
     add_cursor_coords_control(panel_id = "my_panel")
 })

 observe({
   req(input$map_one_loaded, input$map_two_loaded)
   mapProxy("map_one") |>
     remove_cursor_coords_control()
   mapProxy("map_two") |>
     remove_cursor_coords_control(panel_id = "my_panel")
 }) |>
   bindEvent(input$remove_control)
}
}
```

`remove_custom_control` *Remove a custom control from the map*

Description

Remove a custom control from the map

Usage

```
remove_custom_control(proxy, control_id, panel_id = NULL)
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
control_id	The ID of the custom control to remove.
panel_id	Optional. If provided, removes the control from the specified control panel. If `NULL`, removes the standalone custom control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   fluidRow(
     column(6, mapOutput("map_one")),
     column(6, mapOutput("map_two"))
   ),
   actionButton("remove_control", "Remove control")
 )
)
server <- function(input, output, session) {
 output$map_one <- renderMap({
   map() |>
     add_custom_control(
       id = "custom_control",
       html = "<p>I am a custom control</p>"
     )
 })

 output$map_two <- renderMap({
   map() |>
     add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
     add_custom_control(
       id = "custom_control",
       html = "<p>I am a custom control</p>",
       panel_id = "my_panel"
     )
 })

 observe({
   req(input$map_one_loaded, input$map_two_loaded)
   mapProxy("map_one") |>
     remove_custom_control("custom_control")
   mapProxy("map_two") |>
     remove_custom_control("custom_control", panel_id = "my_panel")
 }) |>
```

```
   bindEvent(input$remove_control)
}
}
```

`remove_draw_control` *Remove the draw control from the map*

Description

Remove the draw control from the map

Usage

```
remove_draw_control(proxy, panel_id = NULL)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`panel_id`	Optional. If provided, removes the draw control from the specified control panel. If NULL, removes the standalone draw control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_draw_control", "Remove draw control")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
    add_draw_control()
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_draw_control()
 }) |>
   bindEvent(input$remove_draw_control)
}
}
```

`remove_layer_selector_control` *Remove the layer selector control from the map*

Description

Remove the layer selector control from the map

Usage

```
remove_layer_selector_control(proxy, panel_id = NULL)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`panel_id`	Optional. If provided, removes the layer selector control from the specified control panel. If `NULL`, removes the standalone layer selector control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
# Load libraries
library(shiny)
library(toro)
library(spData)
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)

nz_data <- spData::nz_height |>
  sf::st_transform(4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_control", "Remove layer selector control")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_circle_layer(
```

```
      id = "quakes",
      source = quakes_data
    ) |>
    add_circle_layer(
      id = "nz_elevation",
      source = nz_data
    ) |>
    add_layer_selector_control(
      layer_ids = c("quakes", "nz_elevation"),
      labels = c("quakes" = "Earthquakes", "nz_elevation" = "NZ Elevation")
    )
})

observe({
  req(input$map_loaded)
  mapProxy("map") |>
    remove_layer_selector_control()
}) |>
  bindEvent(input$remove_control)
}
}
```

`remove_route` *Remove an animation route from a toro map*

Description

Remove an animation route from a toro map

Usage

```
remove_route(map, route_id, settings = list())
```

Arguments

`map`	A toro map proxy object.
`route_id`	A unique identifier for the route.
`settings`	A list of settings for removing the route.

Value

The updated map proxy object.

Examples

```
if(interactive()){
library(shiny)
library(toro)
library(sf)
```

```
line_data <- sf::st_sf(
  id = 1,
  geometry = sf::st_sfc(
    sf::st_linestring(
      cbind(c(172.2041, 163.9383), c(-32.56960, -46.43999))
    ),
    crs = 4326
  )
)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("play_route", "Play Route Animation"),
   actionButton("pause_route", "Pause Route Animation"),
   actionButton("remove_route", "Remove Route")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_route(route_id = "route_line", points = line_data)
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     play_route(route_id = "route_line")
 }) |>
   bindEvent(input$play_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     pause_route(route_id = "route_line")
 }) |>
   bindEvent(input$pause_route)

  observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_route(route_id = "route_line")
 }) |>
   bindEvent(input$remove_route)
}
}
```

`remove_speed_control` *Remove the speed control from the map*

Description

Remove the speed control from the map

Usage

```
remove_speed_control(proxy, panel_id = NULL)
```

Arguments

proxy	The map proxy object created by mapProxy().
panel_id	Optional. If provided, removes the speed control from the specified control panel. If NULL, removes the standalone speed control.

Value

The map proxy object for chaining.

Examples

```
# Add to a map
map() |>
 add_speed_control()

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_speed_control(panel_id = "my_panel")
```

---

remove_tile_selector_control

Remove the tile selector control from the map

Description

Remove the tile selector control from the map

Usage

```
remove_tile_selector_control(proxy, panel_id = NULL)
```

Arguments

proxy	The map proxy object created by mapProxy().
panel_id	Optional. If provided, removes the tile selector control from the specified control panel. If NULL, removes the standalone tile selector control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
# Load libraries
library(shiny)
library(toro)

all_tiles <- get_tile_options()

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_control", "Remove tile selector control")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map(loadedTiles = all_tiles) |>
     add_tile_selector_control(available_tiles = all_tiles)
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_tile_selector_control()
 }) |>
   bindEvent(input$remove_control)
}
}
```

---

remove_timeline_control

Remove the timeline control from the map

Description

Remove the timeline control from the map

Usage

```
remove_timeline_control(proxy, panel_id = NULL)
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
panel_id	Optional. If provided, removes the timeline control from the specified control panel. If NULL, removes the standalone timeline control.

Value

The map proxy object for chaining.

Examples

```
# Add to a map
map() |>
 add_timeline_control()

# Add to a control panel
map() |>
 add_control_panel(panel_id = "my_panel", title = "Map Settings") |>
 add_timeline_control(panel_id = "my_panel")
```

---

remove_visibility_toggle

Remove a visibility toggle control from the map

Description

Remove a visibility toggle control from the map

Usage

```
remove_visibility_toggle(proxy, layer_id, panel_id = NULL)
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
layer_id	The ID of the layer whose visibility toggle control to remove.
panel_id	Optional. If provided, removes the control from the specified control panel.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
# Load libraries
library(shiny)
library(toro)
library(sf)

# Prepare data
data(quakes)
quakes_data <- quakes |>
 sf::st_as_sf(coords = c("long", "lat"), crs = 4326)
```

```
ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("remove_control", "Remove visibility control")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   add_circle_layer(
     id = "quakes",
     source = quakes_data
   ) |>
   add_visibility_toggle(layer_id = "quakes")
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     remove_visibility_toggle("quakes")
 }) |>
   bindEvent(input$remove_control)
}
}
```

`remove_zoom_control` *Remove the zoom control from the map*

Description

Remove the zoom control from the map

Usage

```
remove_zoom_control(proxy, panel_id = NULL)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`panel_id`	Optional. If provided, removes the zoom control from the specified control panel. If `NULL`, removes the standalone zoom control.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   checkboxInput("has_zoom_controls", "Remove Zoom Controls", value = TRUE)
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_zoom_control()
 })

 observe({
   req(input$map_loaded)
   if (input$has_zoom_controls == TRUE) {
     mapProxy("map") |>
       add_zoom_control()
   } else {
     mapProxy("map") |>
       remove_zoom_control()
   }
 }) |>
   bindEvent(input$has_zoom_controls)
}
}
```

`renderMap` *Render a MapLibre GL map in Shiny*

Description

Render a MapLibre GL map in Shiny

Usage

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

Arguments

`expr`	An expression that generates a 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.

Value

A rendered MapLibre GL map for use in a Shiny server.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map()
 })
}
}
```

`save_map_html` *Save map as standalone HTML file*

Description

This function saves a map widget as a self-contained HTML file that can be opened in any web browser.

Usage

```
save_map_html(map, filepath, title = "Toro Map", selfcontained = TRUE, ...)
```

Arguments

`map`	A map object created by `map()`.
`filepath`	The file path to save the HTML file (should end with .html).
`title`	The title for the HTML page. Default is "Toro Map".
`selfcontained`	Whether to create a self-contained HTML file. Default is TRUE.
`...`	Additional arguments passed to `htmlwidgets::saveWidget()`.

Value

The file path of the saved HTML file (invisibly).

Examples

```
# Load library
library(sf)

data <- data.frame(lon = 174.8210, lat = -41.3096) |>
  sf::st_as_sf(coords = c("lon", "lat"), crs = 4326)
# Create and export a map
my_map <- map() |>
  add_circle_layer("epi_circle", source = data)

save_map_html(my_map, file.path(tempdir(), "my_map.html"))
```

---

set_bounds *Set the map bounds*

Description

Set the map bounds

Usage

```
set_bounds(map, bounds, padding = 50, max_zoom = map$maxZoom)
```

Arguments

map	The map or map proxy object.
bounds	One of two formats: • A list of two coordinate pairs: `list(list(lon1, lat1), list(lon2, lat2))` • An `sf` object, which will be converted to a bounding box
padding	The padding around the bounds in pixels. Default is 50.
max_zoom	The maximum zoom level to set. Default is the object's `maxZoom`.

Value

The map or map proxy object for chaining.

Examples

```
# Load libraries
library(toro)
library(spData)
library(sf)

nz_data <- spData::nz_height |>
  sf::st_transform(4326)
```

```
map() |>
  set_bounds(list(list(-79, 43), list(-73, 45)))

map() |>
 set_bounds(bounds = nz_data)
```

`set_layout_property` *Set a layout property for a layer on the map*

Description

Set a layout property for a layer on the map

Usage

```
set_layout_property(proxy, layer_id, property_name, value)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`layer_id`	The ID of the layer to update.
`property_name`	The name of the layout property to set.
`value`	The value to set for the layout property.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   selectInput(
     "text_column",
     "Select Text Column",
     choices = c("depth", "mag", "stations")
   )
 )
)
server <- function(input, output, session) {
```

```
  output$map <- renderMap({
    map() |>
      add_text_layer(
        id = "quakes",
        source = quakes_data,
        layout = get_layout_options(
          "text",
          options = list(
            text_field = get_column("depth")
          )
        )
      )
  })

  observe({
    req(input$map_loaded)
    mapProxy("map") |>
      set_layout_property(
        layer_id = "quakes",
        property_name = "text-field",
        value = get_column(input$text_column)
      )
  }) |>
    bindEvent(input$text_column)
}
}
```

`set_paint_property` *Set a paint property for a layer on the map*

Description

Set a paint property for a layer on the map

Usage

```
set_paint_property(proxy, layer_id, property_name, value)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`layer_id`	The ID of the layer to update.
`property_name`	The name of the paint property to set.
`value`	The value to set for the paint property.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   selectInput(
     "colour",
     "Select Tile Layer",
     choices = c(
       "red",
       "orange",
       "yellow",
       "green",
       "blue",
       "indigo",
       "violet"
     )
   )
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_circle_layer(id = "quakes", source = quakes_data)
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     set_paint_property(
       layer_id = "quakes",
       property_name = "circle-color",
       value = input$colour
     )
 }) |>
   bindEvent(input$colour)
}
}
```

`set_source_data` *Set data for a source on the map*

Description

Set data for a source on the map

Usage

```
set_source_data(proxy, source_id, data, type = "geojson")
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
source_id	The ID of the source to update.
data	The data for the source, typically in GeoJSON format.
type	The type of the source. Default is `"geojson"`. Other options include `"vector"` or `"raster"`.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("btn", "Update Source Data")
 )
)
server <- function(input, output, session) {
 get_random_points <- function(data, n = 10) {
   random_rows <- runif(n, min = 1, max = nrow(data))
   return(data[random_rows, ])
 }

 output$map <- renderMap({
   map() |>
     add_symbol_layer(
       id = "quakes",
       source = get_random_points(quakes_data)
     )
   })
 observe({
   mapProxy("map") |>
```

```
      set_source_data(
        source_id = "source-quakes",
        data = get_random_points(quakes_data)
      )
  }) |>
  bindEvent(input$btn)
}
}
```

`set_tile_layer` *Set the tile layer for the map*

Description

Set the tile layer for the map

Usage

```
set_tile_layer(map, tiles)
```

Arguments

`map`	The map or map proxy object.
`tiles`	A character vector of tile layer names. Options include values returned from `get_tile_options()`.

Value

The map or map proxy object for chaining.

See Also

`get_tile_options()` for retrieving all tile options.

Examples

```
if(interactive()){
library(shiny)
library(toro)

all_tiles <- get_tile_options()

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   selectInput("tile_layer", "Select Tile Layer", choices = all_tiles)
 )
)
server <- function(input, output, session) {
```

```
  output$map <- renderMap({
    map(loadedTiles = all_tiles)
  })

  observe({
    mapProxy("map") |>
      set_tile_layer(tiles = input$tile_layer)
  }) |>
    bindEvent(input$tile_layer)
}
}
```

`set_zoom` *Set the map zoom level*

Description

Set the map zoom level

Usage

```
set_zoom(map, zoom)
```

Arguments

`map`	The map or map proxy object.
`zoom`	The zoom level to set. Default is 2.

Value

The map or map proxy object for chaining.

Examples

```
map() |>
 set_zoom(5)
```

show_layer *Show a previously hidden layer on the map*

Description

Show a previously hidden layer on the map

Usage

```
show_layer(proxy, layer_id)
```

Arguments

proxy	The map proxy object created by `mapProxy()`.
layer_id	The ID of the layer to show.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   actionButton("show_layer", "Show Layer"),
   actionButton("hide_layer", "Hide Layer")
 )
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_circle_layer(
       id = "quakes",
       source = quakes_data
     )
 })

 observe({
   mapProxy("map") |>
     show_layer(layer_id = "quakes")
```

```
 }) |>
   bindEvent(input$show_layer)

 observe({
   mapProxy("map") |>
     hide_layer(layer_id = "quakes")
 }) |>
   bindEvent(input$hide_layer)
}
}
```

`toggle_clustering` *Toggle clustering for a layer on the map*

Description

Toggle clustering for a layer on the map

Usage

```
toggle_clustering(proxy, layer_id, cluster = FALSE)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`layer_id`	The ID of the layer to toggle clustering for.
`cluster`	Whether to enable clustering. Default is `FALSE`.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(sf)
library(toro)

data(quakes)
quakes_data <- sf::st_as_sf(quakes, coords = c("long", "lat"), crs = 4326)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   checkboxInput("toggle_cluster", "Toggle Clustering", value = TRUE)
 )
)
server <- function(input, output, session) {
```

```
output$map <- renderMap({
  map() |>
    add_symbol_layer(
      id = "quakes",
      source = quakes_data,
      can_cluster = TRUE
    )
})

observe({
  mapProxy("map") |>
    toggle_clustering(layer_id = "quakes", cluster = input$toggle_cluster)
}) |>
  bindEvent(input$toggle_cluster)
}
}
```

`toggle_control` *Toggle the visibility of a control on the map*

Description

Toggle the visibility of a control on the map

Usage

```
toggle_control(proxy, control_id, show = TRUE)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`control_id`	The ID of the control to toggle.
`show`	Logical indicating whether to show or hide the control. Default is `TRUE`.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
   mapOutput("map"),
   checkboxInput("show_controls", "Show controls", value = TRUE)
 )
```

```
)
server <- function(input, output, session) {
 output$map <- renderMap({
   map() |>
     add_zoom_control() |>
     add_custom_control(
       id = "custom_control",
       html = "<p>I am a custom control</p>"
     )
 })

 observe({
   req(input$map_loaded)
   mapProxy("map") |>
     toggle_control("zoom_control", show = input$show_controls) |>
     toggle_control("custom_control", show = input$show_controls)
 }) |>
   bindEvent(input$show_controls)
}
}
```

`toggle_lat_lng_grid` *Show/hide the latitude and longitude grid on the map*

Description

Show/hide the latitude and longitude grid on the map

Usage

```
toggle_lat_lng_grid(proxy, show = TRUE)
```

Arguments

`proxy`	The map proxy object created by `mapProxy()`.
`show`	Logical indicating whether to show or hide the grid. Default is `TRUE`.

Value

The map proxy object for chaining.

Examples

```
if(interactive()){
library(shiny)
library(toro)

ui <- fluidPage(
 tagList(
```

```
    mapOutput("map"),
    actionButton("show_grid", "Show Grid"),
    actionButton("hide_grid", "Hide Grid")
  )
)
server <- function(input, output, session) {

  output$map <- renderMap({
    map() |>
      add_lat_lng_grid()
  })

  observe({
    mapProxy("map") |>
      toggle_lat_lng_grid(show = TRUE)
  }) |>
    bindEvent(input$show_grid)

  observe({
    mapProxy("map") |>
      toggle_lat_lng_grid(show = FALSE)
   }) |>
     bindEvent(input$hide_grid)
}
}
```

Index