Package: arcgisutils (via r-universe)

July 11, 2024

Title ArcGIS Utility Functions

Version 0.3.0.9000

Description Developer oriented utility functions designed to be used as the building blocks of R packages that work with ArcGIS Location Services. It provides functionality for authorization, Esri JSON construction and parsing, as well as other utilities pertaining to geometry and Esri type conversions. To support 'ArcGIS Pro' users, authorization can be done via 'arcgisbinding'. Installation instructions for 'arcgisbinding' can be found at <https://r.esri.com/r-bridge-site/arcgisbinding/installing-arcgisbinding.html>.

License Apache License (>= 2)

URL https://github.com/R-ArcGIS/arcgisutils, https://r.esri.com/arcgisutils/

BugReports https://github.com/R-ArcGIS/arcgisutils/issues

Imports cli, dbplyr, httr2 (>= 1.0.0), RcppSimdJson, rlang, sf, utils

Suggests arcgisbinding, collapse (>= 2.0.0), data.table, vctrs, testthat (>= 3.0.0), jsonify

Config/testthat/edition 3

Encoding UTF-8

Language en

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Config/rextendr/version 0.3.1.9000

SystemRequirements Cargo (Rust's package manager), rustc

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

RemoteUrl https://github.com/r-arcgis/arcgisutils

RemoteRef HEAD

RemoteSha d9dcac8117a8932d66f0dcb3f6c6822f44459f47

Contents

`arc_agent` *Set user-agent for arcgisutils*

Description

Override the default user-agent set by httr2 to indicate that a request came from arcgisutils.

Usage

```
arc_agent(req)
```

Arguments

`req`	an httr2 request

Value

an httr2 request object

Examples

```
req <- httr2::request("http://example.com")
arc_agent(req)
```

arc_base_req *Generate base request*

Description

This function takes a url and creates a basic httr2 request that adds the user-agent and adds an authorization token to the `X-Esri-Authorization` header.

Usage

```
arc_base_req(
  url,
  token = NULL,
  path = NULL,
  query = NULL,
  error_call = rlang::caller_env()
)
```

Arguments

`url`	a valid url that is passed to `httr2::request()`
`token`	an object of class `httr2_token` as generated by `auth_code()` or related function
`path`	a character vector of paths to be appended to url using `httr2::req_url_path_append()`
`query`	a named vector or named list of query parameters to be appended to the url using `httr2::req_url_query()`
`error_call`	the caller environment to be used when propagating errors.

Examples

```
arc_base_req("https://arcgis.com")
```

arc_host *Determines Portal Host*

Description

Returns a scalar character indicating the host to make requests to.

Usage

```
arc_host()
```

Details

By default, the host is ArcGIS Online <https://www.arcgis.com>. If the environment variable ARCGIS_HOST is set, it will be returned.

Value

A scalar character, "https://www.arcgis.com" by default.

Examples

```
arc_host()
```

---

arc_self_meta *Access the Self Resource*

Description

The function returns the /self resource from the ArcGIS REST API. The /self endpoint returns the view of the portal as seen by the current user, whether anonymous or signed in.

Usage

```
arc_self_meta(token = arc_token(), error_call = rlang::current_call())
```

Arguments

token	an object of class httr2_token as generated by auth_code() or related function
error_call	the caller environment to be used when propagating errors.

Details

See the endpoint documentation for more details.

The Portal Self response can vary based on whether it's called by a user, an app, or both.

The response includes user and appinfo properties, and the variations in responses are primarily related to these two properties. As the names indicate, the user property includes information about the user making the call, and the appinfo property includes information pertaining to the app that made the call.

Value

A named list.

Examples

```
## Not run:
set_arc_token(auth_code())
self <- arc_self_meta()
names(self)

## End(Not run)
```

`arc_token` *Manage authorization tokens*

Description

These functions are used to set, fetch, and check authorization tokens.

Usage

```
arc_token(token = "ARCGIS_TOKEN")

set_arc_token(token, ...)

unset_arc_token(token = NULL)

obj_check_token(token, call = rlang::caller_env())

check_token_has_user(token, call = rlang::caller_env())
```

Arguments

`token`	for `arc_token()`, the name of a token to fetch. For `set_arc_token()`, it is an `httr2_token` that will be set. For `unset_arc_token()`, a character vector of token names to be unset.
`...`	named arguments to set `httr2_token`. Must be valid names and must be an `httr2_token`.
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Details

It is possible to have multiple authorization tokens in one session. These functions assist you in managing them.

`arc_token()` is used to fetch tokens by name. The default token is `ARCGIS_TOKEN`. However, they can be any valid character scalar. `set_arc_token()` will create store a token with the name `ARCGIS_TOKEN`. However, you can alternatively set the tokens by name using a key-value pair. The key is what you would pass to `arc_token()` to fetch the `httr2_token` object. To remove a token that has been set, use `unset_arc_token()`.

`obj_check_token()` is a developer oriented function that can be used to check if an object is indeed an `httr2_token`. To check if a token has expired, `validate_or_refresh_token()` will do so.

`check_token_has_user()` is a developer oriented function that checks to see if a token has a `username` field associated with it.

For developers:

`set_arc_token()` uses a package level environment to store the tokens. The tokens are fetched from the environment using `arc_token()`.

Examples

```
# create fake tokens
token_a <- httr2::oauth_token("1234", arcgis_host = arc_host())
token_b <- httr2::oauth_token("abcd", arcgis_host = arc_host())

# set token to the default location
set_arc_token(token_a)

# fetch token from the default location
arc_token()

# set token by name
set_arc_token(org_a = token_a, org_b = token_b)

# fetch token by name
arc_token("org_a")
arc_token("org_b")

# unset tokens
unset_arc_token()
unset_arc_token(c("org_a", "org_b"))
```

---

as_esri_geometry *Create Esri JSON Geometry Objects*

Description

`as_esri_geometry()` converts an `sfg` object to a EsriJSON Geometry object as a string.

Usage

```
as_esri_geometry(x, crs = NULL, call = rlang::caller_env())
```

Arguments

x	an object of class sfg. Must be one of "POINT", "MULTIPOINT", "LINESTRING", "MULTILINESTRING", "POLYGON", or "MULTIPOLYGON".
crs	the coordinate reference system. It must be interpretable by sf::st_crs().
call	The execution environment of a currently running function, e.g. call = caller_env(). The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply call when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Details

See as_featureset() and as_features() for converting sfc and sf objects into EsriJSON.

Value

a scalar string

References

API Reference

Examples

```
library(sf)
# POINT
# create sfg points
xy <- st_point(c(1, 2))
xyz <- st_point(c(1, 2, 3))
xym <- st_point(c(1, 2, 3), dim = "XYM")
xyzm <- st_point(c(1, 2, 3, 4))

as_esri_geometry(xy)
as_esri_geometry(xyz)
as_esri_geometry(xym)
as_esri_geometry(xyzm)

# MULTIPOINT
# vector to create matrix points
set.seed(0)
x <- rnorm(12)
```

```
xy <- st_multipoint(matrix(x, ncol = 2))
xyz <- st_multipoint(matrix(x, ncol = 3))
xym <- st_multipoint(matrix(x, ncol = 3), dim = "XYM")
xyzm <- st_multipoint(matrix(x, ncol = 4), dim = "XYM")

as_esri_geometry(xy)
as_esri_geometry(xyz)
as_esri_geometry(xym)
as_esri_geometry(xyzm)

# LINESTRING
xy <- st_linestring(matrix(x, ncol = 2))
xyz <- st_linestring(matrix(x, ncol = 3))
xym <- st_linestring(matrix(x, ncol = 3), dim = "XYM")
xyzm <- st_linestring(matrix(x, ncol = 4), dim = "XYM")

as_esri_geometry(xy)
as_esri_geometry(xyz)
as_esri_geometry(xym)
as_esri_geometry(xyzm)

# MULTILINESTRING
as_esri_geometry(st_multilinestring(list(xy, xy)))
as_esri_geometry(st_multilinestring(list(xyz, xyz)))
as_esri_geometry(st_multilinestring(list(xym, xym)))
as_esri_geometry(st_multilinestring(list(xyzm, xyzm)))

# POLYGON
coords <- rbind(
  c(0, 0, 0, 1),
  c(0, 1, 0, 1),
  c(1, 1, 1, 1),
  c(1, 0, 1, 1),
  c(0, 0, 0, 1)
)

xy <- st_polygon(list(coords[, 1:2]))
xyz <- st_polygon(list(coords[, 1:3]))
xym <- st_polygon(list(coords[, 1:3]), dim = "XYM")
xyzm <- st_polygon(list(coords))

as_esri_geometry(xy)
as_esri_geometry(xyz)
as_esri_geometry(xym)
as_esri_geometry(xyzm)

# MULTIPOLYGON
as_esri_geometry(st_multipolygon(list(xy, xy)))
as_esri_geometry(st_multipolygon(list(xyz, xyz)))
as_esri_geometry(st_multipolygon(list(xym, xym)))
as_esri_geometry(st_multipolygon(list(xyzm, xyzm)))
```

`as_extent` *Convert an object to an extent*

Description

Given an sf or sfc object create a list that represents the extent of the object. The result of this function can be parsed directly into json using `jsonify::to_json(x, unbox = TRUE)` or included into a list as the extent component that will be eventually converted into json using the above function.

Usage

```
as_extent(x, crs = sf::st_crs(x), call = rlang::caller_env())
```

Arguments

`x`	an sf or sfc object
`crs`	the CRS of the object. Must be parsable by `sf::st_crs()`
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Value

An extent json object. Use `jsonify::to_json(x, unbox = TRUE)` to convert to json.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
as_extent(nc)
```

`as_features` *Create Esri Features*

Description

These functions create an array of Esri Feature objects. Each feature consists of a geometry and attribute field. The result of `as_esri_features()` is a JSON array of Features whereas `as_features()` is a list that represents the same JSON array. Using `jsonify::to_json(as_features(x), unbox = TRUE)` will result in the same JSON array.

Usage

```
as_features(x, crs = sf::st_crs(x), call = rlang::caller_env())

as_esri_features(x, crs = sf::st_crs(x), call = rlang::caller_env())
```

Arguments

`x`	an object of class `sf`, `data.frame`, or `sfc`.
`crs`	the coordinate reference system. It must be interpretable by `sf::st_crs()`.
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Value

Either a scalar string or a named list.

References

API Reference

Examples

```
library(sf)
# POINT
# create sfg points
xy <- st_sfc(st_point(c(1, 2)))
xyz <- st_sfc(st_point(c(1, 2, 3)))
xym <- st_sfc(st_point(c(1, 2, 3), dim = "XYM"))
```

```
as_esri_features(xy)
as_esri_features(xyz)
as_esri_features(xym)

# MULTIPOINT
# vector to create matrix points
set.seed(0)
x <- rnorm(12)

xy <- st_sfc(st_multipoint(matrix(x, ncol = 2)))
xyz <- st_sfc(st_multipoint(matrix(x, ncol = 3)))
xym <- st_sfc(st_multipoint(matrix(x, ncol = 3), dim = "XYM"))

as_esri_features(xy)
as_esri_features(xyz)
as_esri_features(xym)

# LINESTRING
xy <- st_sfc(st_linestring(matrix(x, ncol = 2)))
xyz <- st_sfc(st_linestring(matrix(x, ncol = 3)))
xym <- st_sfc(st_linestring(matrix(x, ncol = 3), dim = "XYM"))

as_esri_features(xy)
as_esri_features(xyz)
as_esri_features(xym)

# MULTILINESTRING
as_esri_features(st_sfc(st_multilinestring(list(xy[[1]], xy[[1]]))))
as_esri_features(st_sfc(st_multilinestring(list(xyz[[1]], xyz[[1]]))))
as_esri_features(st_sfc(st_multilinestring(list(xym[[1]], xym[[1]]))))

# POLYGON
coords <- rbind(
  c(0, 0, 0, 1),
  c(0, 1, 0, 1),
  c(1, 1, 1, 1),
  c(1, 0, 1, 1),
  c(0, 0, 0, 1)
)

xy <- st_sfc(st_polygon(list(coords[, 1:2])))
xyz <- st_sfc(st_polygon(list(coords[, 1:3])))
xym <- st_sfc(st_polygon(list(coords[, 1:3]), dim = "XYM"))

as_esri_features(xy)
as_esri_features(xyz)
as_esri_features(xym)

# MULTIPOLYGON
as_esri_features(st_sfc(st_multipolygon(list(xy[[1]], xy[[1]]))))
as_esri_features(st_sfc(st_multipolygon(list(xyz[[1]], xyz[[1]]))))
as_esri_features(st_sfc(st_multipolygon(list(xym[[1]], xym[[1]]))))
```

as_featureset *Create Esri FeatureSet Objects*

Description

These functions create an Esri FeatureSet object. A FeatureSet contains an inner array of features as well as additional metadata about the the collection such as the geometry type, spatial reference, and object ID field.

Usage

```
as_featureset(x, crs = sf::st_crs(x), call = rlang::caller_env())

as_esri_featureset(x, crs = sf::st_crs(x), call = rlang::caller_env())
```

Arguments

x	an object of class sf, data.frame, or sfc.
crs	the coordinate reference system. It must be interpretable by sf::st_crs().
call	The execution environment of a currently running function, e.g. call = caller_env(). The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply call when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

References

API Reference

Examples

```
library(sf)
# POINT
# create sfg points
xy <- st_sfc(st_point(c(1, 2)))
xyz <- st_sfc(st_point(c(1, 2, 3)))
xym <- st_sfc(st_point(c(1, 2, 3), dim = "XYM"))

as_esri_featureset(xy)
as_esri_featureset(xyz)
as_esri_featureset(xym)

# MULTIPOINT
```

```
# vector to create matrix points
set.seed(0)
x <- rnorm(12)

xy <- st_sfc(st_multipoint(matrix(x, ncol = 2)))
xyz <- st_sfc(st_multipoint(matrix(x, ncol = 3)))
xym <- st_sfc(st_multipoint(matrix(x, ncol = 3), dim = "XYM"))

as_esri_featureset(xy)
as_esri_featureset(xyz)
as_esri_featureset(xym)

# LINESTRING
xy <- st_sfc(st_linestring(matrix(x, ncol = 2)))
xyz <- st_sfc(st_linestring(matrix(x, ncol = 3)))
xym <- st_sfc(st_linestring(matrix(x, ncol = 3), dim = "XYM"))

as_esri_featureset(xy)
as_esri_featureset(xyz)
as_esri_featureset(xym)

# MULTILINESTRING
as_esri_featureset(st_sfc(st_multilinestring(list(xy[[1]], xy[[1]]))))
as_esri_featureset(st_sfc(st_multilinestring(list(xyz[[1]], xyz[[1]]))))
as_esri_featureset(st_sfc(st_multilinestring(list(xym[[1]], xym[[1]]))))

# POLYGON
coords <- rbind(
  c(0, 0, 0, 1),
  c(0, 1, 0, 1),
  c(1, 1, 1, 1),
  c(1, 0, 1, 1),
  c(0, 0, 0, 1)
)

xy <- st_sfc(st_polygon(list(coords[, 1:2])))
xyz <- st_sfc(st_polygon(list(coords[, 1:3])))
xym <- st_sfc(st_polygon(list(coords[, 1:3]), dim = "XYM"))

as_esri_featureset(xy)
as_esri_featureset(xyz)
as_esri_featureset(xym)

# MULTIPOLYGON
as_esri_featureset(st_sfc(st_multipolygon(list(xy[[1]], xy[[1]]))))
as_esri_featureset(st_sfc(st_multipolygon(list(xyz[[1]], xyz[[1]]))))
as_esri_featureset(st_sfc(st_multipolygon(list(xym[[1]], xym[[1]]))))
```

---

as_layer *Create Esri layer objects*

Description

These functions are used to generate list objects that can be converted into json objects that are used in REST API requests. Notably they are used for adding R objects as items to a portal.

Usage

```
as_layer(
  x,
  name,
  title,
 layer_definition = as_layer_definition(x, name, "object_id", infer_esri_type(x)),
  id = NULL,
  layer_url = NULL,
  legend_url = NULL,
  popup_info = NULL,
  call = rlang::caller_env()
)

as_layer_definition(
  x,
  name,
  object_id_field,
  fields = infer_esri_type(x),
  display_field = NULL,
  drawing_info = NULL,
  has_attachments = FALSE,
  max_scale = 0,
  min_scale = 0,
  templates = NULL,
  type_id_field = NULL,
  types = NULL,
  call = rlang::caller_env()
)

as_feature_collection(
  layers = list(),
  show_legend = TRUE,
  call = rlang::caller_env()
)
```

Arguments

x	an object of class `data.frame`. This can be an `sf` object or `tibble` or any other subclass of `data.frame`.
name	a scalar character of the name of the layer. Must be unique.
title	A user-friendly string title for the layer that can be used in a table of contents.

`layer_definition`
: a layer definition list as created by `as_layer_definition()`. A default is derived from `x` and the `name` object.

`id`
: A number indicating the index position of the layer in the WMS or map service.

`layer_url`
: default `NULL`. A string URL to a service that should be used for all queries against the layer. Used with hosted tiled map services on ArcGIS Online when there is an associated feature service that allows for queries.

`legend_url`
: default `NULL`. A string URL to a legend graphic for the layer. Used with WMS layers. The URL usually contains a GetLegendGraphic request.

`popup_info`
: default `NULL`. A list that can be converted into a popupInfo object defining the pop-up window content for the layer. There is no helper for popupInfo objects.

`call`
: The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error.
  You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message.
  Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display.
  For more information about error calls, see Including function calls in error messages.

`object_id_field`
: a scalar character vector indicating the name of the object ID field in the dataset.

`fields`
: a data.frame describing the fields in `x`. These values are inferred by default via `infer_esri_type()`.

`display_field`
: default `NULL`. A scalar character containing the name of the field that best summarizes the feature. Values from this field are used by default as the titles for pop-up windows.

`drawing_info`
: default `NULL`. See REST documentation in details for more. There are no helpers or validators for `drawingInfo` objects.

`has_attachments`
: default `FALSE`.

`max_scale`
: default `NULL`. A number representing the maximum scale at which the layer definition will be applied. The number is the scale's denominator; thus, a value of 2400 represents a scale of 1/2,400. A value of 0 indicates that the layer definition will be applied regardless of how far you zoom in.

`min_scale`
: default `NULL`. A number representing the minimum scale at which the layer definition will be applied.

`templates`
: default `NULL`. See REST documentation in details for more.

`type_id_field`
: default `NULL`. See REST documentation in details for more.

`types`
: An array of type objects available for the dataset. This is used when the `type_id_field` is populated. NOTE there are no helper functions to create type objects. Any type list objects must match the json structure when passed to `jsonify::to_json(x, unbox = TRUE)`.

`layers`
: a list of layers as created by `as_layer()`.

`show_legend`
: default `FALSE`. Logical scalar indicating if this layer should be shown in the legend in client applications.

Details

A `featureCollection` defines a layer of features that will be stored on a web map. It consists of an array of `layers`. The `layer` contains the features (attributes and geometries) as a `featureSet` (see `as_esri_featureset()`) and additional metadata which is stored in the `layerDefinition`object. The `layerDefinition` most importantly documents the fields in the object, the object ID, and additional metadata such as name, title, and display scale.

Additional documentation for these json object:

- `layer`
- `layerDefinition`
- `featureCollection`

Value

A list object containing the required fields for each respective json type. The results can be converted to json using `jsonify::to_json(x, unbox = TRUE)`

Examples

```
ld <- as_layer_definition(iris, "iris", "objectID")
l <- as_layer(iris, "iris name", "Iris Title")
fc <- as_feature_collection(layers = list(l))
```

---

auth_code *Authorization*

Description

Authorize your R session to connect to an ArcGIS Portal. See details.

Usage

```
auth_code(client = Sys.getenv("ARCGIS_CLIENT"), host = arc_host())

auth_client(
  client = Sys.getenv("ARCGIS_CLIENT"),
  secret = Sys.getenv("ARCGIS_SECRET"),
  host = arc_host(),
  expiration = 120
)

auth_binding()

auth_user(
  username = Sys.getenv("ARCGIS_USER"),
  password = Sys.getenv("ARCGIS_PASSWORD"),
  host = arc_host(),
```

```
  expiration = 60
)

auth_key(api_key = Sys.getenv("ARCGIS_API_KEY"), host = arc_host())

refresh_token(token, client = Sys.getenv("ARCGIS_CLIENT"), host = arc_host())

validate_or_refresh_token(
  token,
  client = Sys.getenv("ARCGIS_CLIENT"),
  host = arc_host(),
  refresh_threshold = 0,
  call = rlang::caller_env()
)
```

Arguments

client	an OAuth 2.0 developer application client ID. By default uses the environment variable ARCGIS_CLIENT.
host	default "https://www.arcgis.com"
secret	an OAuth 2.0 developer application secret. By default uses the environment variable ARCGIS_SECRET.
expiration	the duration of the token in minutes.
username	default Sys.getenv("ARCGIS_USER"). Your username to login. **Do not** hard code this value.
password	default Sys.getenv("ARCGIS_PASSWORD"). Your password to login. **Do not** hard code this value.
api_key	default Sys.getenv("ARCGIS_API_KEY"). A character scalar of an ArcGIS Developer API key.
token	an httr2_token as created by auth_code() or similar
refresh_threshold	default 0. If token expiry is within this threshold (in seconds) the token will be refreshed only if a refresh_token is available. Token refreshing is only possible with auth_code() flow.
call	The execution environment of a currently running function, e.g. call = caller_env(). The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply call when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Details

ArcGIS Online and Enterprise Portals utilize OAuth2 authorization via their REST APIs.

- `auth_code()` is the recommend OAuth2 workflow for interactive sessions
- `auth_client()` is the recommended OAuth2 workflow for non-interactive sessions
- `auth_user()` uses legacy username and password authorization using the `generateToken` endpoint. It is only recommended for legacy systems that do not implement OAuth2.
- `auth_binding()` fetches a token from the active portal set by `arcgisbinding`. Uses `arcgisbinding::arc.check_por` to extract the authorization token. Recommended if using arcgisbinding.

Value

an `httr2_token`

Examples

```
## Not run:
auth_code()
auth_client()
auth_user()
auth_key()
auth_binding()

## End(Not run)
```

---

compact *General utility functions*

Description

General utility functions

Usage

```
compact(.x)

a %||% b

check_dots_named(dots, call = rlang::caller_env())
```

Arguments

`.x`	a list
`a`	an R object
`b`	an R object
`dots`	a list collected from dots via `rlang::list2(...)`
`call`	default `rlang::caller_env()`. The caller environment passed to `cli::cli_abort()`

Details

- `compact()` removes any `NULL` list elements
- `%||%` is a special pipe operator that returns `b` if `a` is `NULL`

Value

- `compact()` a list
- `%||%` the first non-null item or `NULL` if both are `NULL`

Examples

```
# remove null elements
compact(list(a = NULL, b = 1))

# if NULL return rhs
NULL %||% 123

# if not NULL return lhs
123 %||% NULL
```

`detect_errors` *Detect errors in parsed json response*

Description

The requests responses from ArcGIS don't return the status code in the response itself but rather from the body in the json. This function checks for the existence of an error. If an error is found, the contents of the error message are bubbled up.

Usage

```
detect_errors(response, error_call = rlang::caller_env())

catch_error(response, error_call = rlang::caller_env())
```

Arguments

`response`	for `detect_errors()`, a list typically from `RcppSimdJson::fparse(httr2::resp_body_string(resp)` For `catch_error()`, the string from `httr2::resp_body_string(resp)`.
`error_call`	default `rlang::caller_env()`. The environment from which to throw the error from.

Value

Nothing. Used for it's side effect. If an error code is encountered in the response an error is thrown with the error code and the error message.

Examples

```
## Not run:
response <- list(
  error = list(
    code = 400L,
    message = "Unable to generate token.",
    details = "Invalid username or password."
  )
)

detect_errors(response)

## End(Not run)
```

`determine_dims` *Determine the dimensions of a geometry object*

Description

Given an sfc or sfg object determine what dimensions are represented.

Usage

```
determine_dims(x)

has_m(x)

has_z(x)
```

Arguments

`x`	an object of class `sfc` or `sfg`

Value

`determine_dims()` returns a scalar character of the value `"xy"`, `"xyz"`, or `"xyzm"` depending on what dimensions are represented.

`has_m()` and `has_z()` returns a logical scalar of `TRUE` or `FALSE` if the geometry has a Z or M dimension.

Examples

```
geo <- sf::st_read(system.file("shape/nc.shp", package="sf"), quiet = TRUE)[["geometry"]]

determine_dims(geo)
has_z(geo)
has_m(geo)
```

`determine_esri_geo_type` *Determine Esri Geometry type*

Description

Takes an `sf` or `sfc` object and returns the appropriate Esri geometry type.

Usage

```
determine_esri_geo_type(x, call = rlang::caller_env())
```

Arguments

`x`	an object of class `data.frame`, `sf`, `sfc`, or `sfg`.
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Details

Geometry type mapping:

- `POINT`: `esriGeometryPoint`
- `MULTIPOINT`: `esriGeometryMultipoint`
- `LINESTRING`: `esriGeometryPolyline`
- `MULTILINESTRING`: `esriGeometryPolyline`
- `POLYGON`: `esriGeometryPolygon`
- `MULTIPOLYGON`: `esriGeometryPolygon`

Value

returns a character scalar of the corresponding Esri geometry type

Examples

```
determine_esri_geo_type(sf::st_point(c(0, 0)))
```

`fetch_layer_metadata` *Retrieve metadata*

Description

Utility functions for feature service metadata.

Usage

```
fetch_layer_metadata(url, token = NULL, call = rlang::caller_env())
```

Arguments

`url`	the url of the item.
`token`	an `httr2_token` from one of the provided `auth_` functions
`call`	default `rlang::caller_env()`. The calling environment passed to `detect_errors()`.

Details

- `fetch_layer_metadata()` given a request, fetches the metadata by setting the query parameter `f=json`

Value

returns a list object

Examples

```
# url is broken into parts to fit within 100 characters to avoid CRAN notes
url_parts <- c(
  "https://services.arcgis.com/P3ePLMYs2RVChkJx/ArcGIS/rest/services",
  "/USA_Counties_Generalized_Boundaries/FeatureServer/0"
)

furl <- paste0(url_parts, collapse = "")
meta <- fetch_layer_metadata(furl)
head(names(meta))
```

infer_esri_type *Esri field type mapping*

Description

Infers Esri field types from R objects.

Usage

```
infer_esri_type(
  .data,
  arg = rlang::caller_arg(.data),
  call = rlang::caller_env()
)

get_ptype(field_type, n = 1, call = rlang::caller_env())

ptype_tbl(fields, n = 0, call = rlang::caller_env())

remote_ptype_tbl(fields, call = rlang::caller_env())
```

Arguments

.data	an object of class data.frame.
arg	An argument name in the current function.
call	The execution environment of a currently running function, e.g. call = caller_env(). The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply call when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.
field_type	a character of a desired Esri field type. See details for more.
n	the number of rows to create in the prototype table
fields	a data.frame containing, at least, the columns type and name. Typically retrieved from the field metadata from a FeatureLayer or Table. Also can use the output of infer_esri_type().

Details

- get_ptype() takes a scalar character containing the Esri field type and returns a prototype of the pertinent R type
- infer_esri_type() takes a data frame-like object and infers the Esri field type from it.

- `remote_ptype_tbl()` takes a data frame of fields as derived from `list_fields()` and creates a lazy table proto type intended to be used with `dbplyr` integration

Field type mapping::

Esri field types are mapped as

- `esriFieldTypeSmallInteger`: integer
- `esriFieldTypeSingle`: double
- `esriFieldTypeGUID`: integer
- `esriFieldTypeOID`: integer
- `esriFieldTypeInteger`: integer
- `esriFieldTypeBigInteger`: double
- `esriFieldTypeDouble`: double
- `esriFieldTypeString`: character
- `esriFieldTypeDate`: date

R types are mapped as

- `double`: esriFieldTypeDouble
- `integer`: esriFieldTypeInteger
- `character`: esriFieldTypeString
- `date`: esriFieldTypeDate
- `raw`: esriFieldTypeBlob

Value

- `get_pytpe()` returns an object of the class of the prototype.
- `ptype_tbl()` takes a `data.frame` with columns `name` and `type` and creates an empty `data.frame` with the corresponding columns and R types
- `remote_ptype_tbl()` provides the results of `ptype_tbl()` as a lazy data frame from the `dbplyr` package.
- `infer_esri_ptype()` returns a `data.frame` with columns `name`, `type`, `alias`, `nullable`, and `editable` columns
  - This resembles that of the `fields` returned by a FeatureService

Examples

```
get_ptype("esriFieldTypeDouble")
inferred <- infer_esri_type(iris)
ptype_tbl(inferred)
```

`is_date` *Date handling*

Description

Esri date fields are represented as milliseconds from the Unix Epoch.

Usage

```
is_date(x, tz)

date_to_ms(x, tz = "UTC")

from_esri_date(x)
```

Arguments

`x`	an object of class `Date` or `POSIXt`. In the case of `is_date()`, any R object.
`tz`	a character string. The time zone specification to be used for the conversion, *if one is required*. System-specific (see time zones), but `""` is the current time zone, and `"GMT"` is UTC (Universal Time, Coordinated). Invalid values are most commonly treated as UTC, on some platforms with a warning.

Details

- `is_date()`: checks if an object is a `Date` or `POSIXt` class object.
- `date_to_ms()` converts a date object to milliseconds from the Unix Epoch in the specified time zone.

Value

- `is_date()` returns a logical scalar
- `date_to_ms()` returns a numeric vector of times in milliseconds from the Unix Epoch in the specified time zone.

Examples

```
today <- Sys.Date()

is_date(today)

date_to_ms(today)
```

`parse_esri_json` *Parse Esri JSON*

Description

Parses an Esri FeatureSet JSON object into an R object. If there is no geometry present, a data.frame is returned. If there is geometry, an sf object is returned.

Usage

```
parse_esri_json(string, ..., call = rlang::caller_env())
```

Arguments

`string`	the raw Esri JSON string.
`...`	additional arguments passed to `RcppSimdJson::fparse`
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Value

A data.frame. If geometry is found, returns an sf object.

Examples

```
esri_json <- '{
    "geometryType": "esriGeometryPolygon",
    "spatialReference": {
        "wkid": 4326
    },
    "hasZ": false,
    "hasM": false,
    "features": [
        {
            "attributes": {
                "id": 1
            },
            "geometry": {
                "rings": [
                    [
                        [0.0, 0.0],
```

```
                        [1.0, 0.0],
                        [1.0, 1.0],
                        [0.0, 1.0],
                        [0.0, 0.0]
                    ]
                ]
            }
        }
    ]
}'

parse_esri_json(esri_json)
```

`rbind_results` *Combine multiple data.frames*

Description

A general function that takes a list of `data.frames` and returns a single and combines them into a single object. It will use the fastest method available. In order this is `collapse::rowbind()`, `data.table::rbindlist()`, `vctrs::list_unchop()`, then `do.call(rbind.data.frame, x)`.

Usage

```
rbind_results(x, call = rlang::current_env(), .ptype = data.frame())
```

Arguments

`x`	a list where each element is a `data.frame` or `NULL`.
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.
`.ptype`	currently unused. Reserved for a future release.

Details

If all items in the list are `data.frames`, then the result will be a `data.frame`. If all elements are an `sf` object, then the result will be an `sf` object. If the items are mixed, the result will be a `data.frame`.

If any items are `NULL`, then an attribute `null_elements` will be attached to the result. The attribute is an integer vector of the indices that were `NULL`.

Value

see details.

Examples

```
x <- head(iris)
res <- rbind_results(list(x, NULL, x))
attr(res, "null_elements")
```

`validate_crs` *Validate CRS object*

Description

Takes a representation of a CRS and ensures that it is a valid one. The CRS is validated using `sf::st_crs()` if it cannot be validated, a null CRS is returned.

Usage

```
validate_crs(crs, arg = rlang::caller_arg(crs), call = rlang::caller_env())
```

Arguments

`crs`	a representation of a coordinate reference system.
`arg`	An argument name in the current function.
`call`	The execution environment of a currently running function, e.g. `call = caller_env()`. The corresponding function call is retrieved and mentioned in error messages as the source of the error. You only need to supply `call` when throwing a condition from a helper function which wouldn't be relevant to mention in the message. Can also be `NULL` or a defused function call to respectively not display any call or hard-code a code to display. For more information about error calls, see Including function calls in error messages.

Details

See `sf::st_crs()` for more details on valid representations.

Value

Returns a list of length 1 with an element named `spatialReference` which is itself a named list.

If the provided CRS returns a valid well-known ID (WKID) `spatialReference` contains a named element called `wkid` which is the integer value of the WKID. If the WKID is not known but the CRS returned is a valid well-known text representation the `wkid` field is `NA` and another field `wkt` contains the valid wkt.

Examples

```
# using epsg code integer or string representation
validate_crs(3857)
validate_crs("EPSG:4326")

# using a custom proj4 string
proj4string <- "+proj=longlat +datum=WGS84 +no_defs"

crs <- validate_crs(proj4string)

# using wkt2 (from above result)
crs <- validate_crs(crs$spatialReference$wkt)
```

Index