Package 'galah'

February 11, 2026

Type Package

Title Biodiversity Data from the GBIF Node Network

Version 2.2.0

Description The Global Biodiversity Information Facility ('GBIF', <https://www.gbif.org>) sources data from an international network of data providers, known as 'nodes'. Several of these nodes - the ``living atlases'' (<https://living-atlases.gbif.org>) - maintain their own web services using software originally developed by the Atlas of Living Australia ('ALA', <https://www.ala.org.au>). 'galah' enables the R community to directly access data and resources hosted by 'GBIF' and its partner nodes.

Depends R (>= 4.3.0)

Imports cli, crayon, dplyr, glue (>= 1.3.2), httr2, jsonlite (>= 0.9.8), lifecycle (>= 1.0.0), potions (>= 0.2.0), purrr, readr, rlang, sf, stringr, tibble, tidyr, tidyselect, utils, xml2

Suggests covr, gt, kableExtra, knitr, magrittr, pkgdown, reactable, rmarkdown, testthat

License MPL-2.0

URL https://galah.ala.org.au/R/

BugReports https://github.com/AtlasOfLivingAustralia/galah-R/issues

Maintainer Martin Westgate <martin.westgate@csiro.au>

LazyLoad yes

VignetteBuilder knitr

RoxygenNote 7.3.3

Encoding UTF-8

Config/testthat/edition 3

NeedsCompilation no

Author Martin Westgate [aut, cre],
Dax Kellie [aut],
Shandiya Balasubramaniam [ctb],
Matilda Stevenson [ctb]

Repository CRAN

Date/Publication 2026-02-11 08:30:02 UTC

Contents

`apply_profile` *Apply a data quality profile*

Description

A 'profile' is a group of filters that are pre-applied by the ALA. Using a data profile allows a query to be filtered quickly to the most relevant or quality-assured data that is fit-for-purpose. For example, the "ALA" profile is designed to exclude lower quality records, whereas other profiles apply filters specific to species distribution modelling (e.g. CDSM).

Note that only one profile can be loaded at a time; if multiple profiles are given, the first valid profile is used.

For more bespoke editing of filters within a profile, use `filter.data_request()`.

Usage

```
apply_profile(.data, ...)

galah_apply_profile(...)
```

Arguments

`.data`	An object of class `data_request`
`...`	a profile name. Should be a `string` - the name or abbreviation of a data quality profile to apply to the query. Valid values can be seen using `show_all(profiles)`

Value

An updated `data_request` with a completed `apply_profile` slot.

See Also

`show_all()` and `search_all()` to look up available data profiles. `filter.data_request()` can be used for more bespoke editing of individual data profile filters.

Examples

```
## Not run:
# Apply a data quality profile to a query
galah_call() |>
  identify("reptilia") |>
  filter(year == 2021) |>
  apply_profile(ALA) |>
  atlas_counts()

## End(Not run)
```

`arrange.data_request` *Order rows using column values*

Description

`arrange.data_request()` arranges rows of a query on the server side, meaning that the query is constructed in such a way that information will be arranged when the query is processed. This only has an effect when used in combination with `count()` and `group_by()`. The benefit of using `arrange()` within a `galah_call()` pipe is that it is sometimes beneficial to choose a non-default order for data to be delivered in, particularly if `slice_head()` is also called.

Usage

```
## S3 method for class 'data_request'
arrange(.data, ...)

## S3 method for class 'metadata_request'
arrange(.data, ...)
```

Arguments

`.data`	An object of class `data_request`
`...`	A variable to arrange the resulting tibble by. Should be one of the variables also listed in `group_by()`.

Value

An amended `data_request` with a completed `arrange` slot.

Examples

```
## Not run:

# Arrange grouped counts by ascending year
galah_call() |>
  identify("Crinia") |>
  filter(year >= 2020) |>
  group_by(year) |>
  arrange(year) |>
  count() |>
  collect()

# Arrange grouped counts by ascending record count
galah_call() |>
  identify("Crinia") |>
  filter(year >= 2020) |>
  group_by(year) |>
  arrange(count) |>
  count() |>
```

```
  collect()

# Arrange grouped counts by descending year
galah_call() |>
  identify("Crinia") |>
  filter(year >= 2020) |>
  group_by(year) |>
  arrange(desc(year)) |>
  count() |>
  collect()

## End(Not run)
```

`atlas_citation` *Generate a citation for occurrence data*

Description

If a `tibble` containing occurrences was generated using galah (either via `collect()` or `atlas_occurrences()`), it will usually contain associated metadata stored in `attributes()` that can be used to build a citation for that dataset. This function simply extracts that information, formats it, then both invisibly returns the formatted citation and prints it to the console.

Usage

```
atlas_citation(data)
```

Arguments

`data`	A `tibble` generated by `atlas_occurrences()` or similar

Value

Invisibly returns a string containing the citation for that dataset. Primarily called for the side-effect of printing this string to the console.

Examples

```
## Not run:
x <- galah_call() |>
    identify("Heleioporus") |>
    filter(year == 2022) |>
    collect()
atlas_citation(x)

## End(Not run)
```

`authenticate` *Set up authentication*

Description

Add an authentication slot to a query. That slot is then used by later code to determine whether to add an OAuth workflow. It is triggered automatically within `capture()` if the `authenticate` argument of `galah_config()` is set to `TRUE`, but only for occurrence queries to the Atlas of Living Australia. **[Experimental]**.

Usage

```
authenticate(.data, cache_disk = FALSE)
```

Arguments

`.data`	An object of class `data_request` or `metadata_request`
`cache_disk`	(logical) Should JWT tokens be cached to disk? Defaults to `FALSE`

Value

An object of the same class as supplied, but with an added `authenticate` slot.

Examples

```
## Not run:
# use `galah_config()` to set for all occurrence queries
galah_config(authenticate = TRUE)

x <- galah_call() |>
  identify("Wollemia nobilis") |>
  collect()

# use in-pipe for more control
x <- galah_call() |>
  identify("Wollemia nobilis") |>
  authenticate() |>
  collect()

## End(Not run)
```

`capture.data_request` *Capture a request*

Description

The first step in evaluating a request is to capture and parse the information it contains. The resulting object has class `prequery` for those requiring further processing or `query` for those that don't. A `prequery` object shows the basic structure of what has been requested by a user in a given `galah_call()`.

Usage

```
capture(x, ...)

## S3 method for class 'data_request'
capture(x, mint_doi = FALSE, ...)

## S3 method for class 'metadata_request'
capture(x, ...)

## S3 method for class 'files_request'
capture(x, thumbnail = FALSE, ...)

## S3 method for class 'list'
capture(x, ...)
```

Arguments

`x`	A `_request` object to convert to a `prequery`.
`...`	Other arguments, currently ignored
`mint_doi`	Logical: should a DOI be minted for this download? Only applies to `type = "occurrences"` when atlas chosen is "ALA".
`thumbnail`	Logical: should thumbnail-size images be returned? Defaults to `FALSE`, indicating full-size images are required.

Details

`galah` uses an object-based pipeline to convert piped requests into valid queries, and to enact those queries with the specified organisation. Typically, requests open with `galah_call()` - though `request_metadata()` and `request_files()` are also valid - and end with `collect()`. Under the hood, the sequence of functions is as follows:

`capture()` → `compound()` → `collapse()` → `compute()` → `collect()`

`capture()` is the first of the `galah_call()` workflow, and it parses the basic structure of a user request, returned as a `prequery` object. A `prequery` object shows what has been requested, before those calls are built by `compound()` and evaluated by `collapse()`. For simple cases, this gives the same result as running `collapse()` while the `run_checks` argument of `galah_config()` is set to `FALSE`, but is slightly faster. In complex cases, it is simply a precursor to `compound()`.

Value

Either an object of class `prequery` when further processing is required; or `query` when it is not. Both classes are structurally identical, being list-like and containing at the following slots:

- `type`: The type of query, serves as a lookup to the corresponding field in `show_all(apis)`
- `url`: Either:
  - a length-1 character giving the API to be queried; or
  - a `tibble()` containing at least the field `url` and optionally others
- `request`: captures the preceeding `_request` object (see `galah_call()`)

See Also

To open a piped query, see `galah_call()`. For alternative operations on `_request` objects, see `compound()`, `collapse()`, `compute()` or `collect()`.

`collapse.data_request` *Generate a query*

Description

Constructs a query so it can be inspected before being sent. `collapse()` can be called at the end of a pipe that begins with `galah_call()` to return the constructed user query generated by the user's data request (a `query` object). Objects of class `data_request` (created using `request_data()`), `metadata_request` (from `request_metadata()`) or `files_request` (from `request_files()`) are all supported.

Usage

```
## S3 method for class 'data_request'
collapse(x, ...)

## S3 method for class 'metadata_request'
collapse(x, ...)

## S3 method for class 'files_request'
collapse(x, ...)

## S3 method for class 'prequery'
collapse(x, ...)

## S3 method for class 'query'
collapse(x, ...)

## S3 method for class 'query_set'
collapse(x, ...)
```

Arguments

x	An object to run `collapse()` on. Classes supported by `galah` include `data_request`, `metadata_request` and `files_request` for building queries; and `prequery`, `query` or `query_set` once constructed (via `capture()` or `compound()`).
...	Arguments passed on to `capture()`.

Details

`galah` uses an object-based pipeline to convert piped requests into valid queries, and to enact those queries with the specified organisation. Typically, requests open with `galah_call()` - though `request_metadata()` and `request_files()` are also valid - and end with `collect()`. Under the hood, the sequence of functions is as follows:

`capture()` → `compound()` → `collapse()` → `compute()` → `collect()`

`collapse()` constructs a complete user query, ready to be sent by `compute()`. Information required to construct a complete user query are provided by `capture()` and `compound()`, preceding functions to parse and combine all required API calls necessary to build a user's query.

Value

An object of class `query`, which is a list-like object containing two or more of the following slots:

- `type`: The type of query, serves as a lookup to the corresponding field in `show_all(apis)`
- `url`: Either:
  - a length-1 character giving the API to be queried; or
  - a `tibble` containing at least the field `url` and optionally others
- `headers`: headers to be sent with the API call
- `body`: body section of the API call
- `options`: options section of the API call
- `request`: captures the supplied `_request` object (see `galah_call()`)

See Also

To open a piped query, see `galah_call()`. For alternative operations on `_request` objects, see `capture()`, `compound()`, `compute()` or `collect()`.

`collect.data_request` *Retrieve a database query*

Description

Retrieve the result of a query from the server. It is the default way to end a piped query that begins with `galah_call()`.

Usage

```
## S3 method for class 'data_request'
collect(x, ..., wait = TRUE, file = NULL)

## S3 method for class 'metadata_request'
collect(x, ...)

## S3 method for class 'files_request'
collect(x, ...)

## S3 method for class 'prequery'
collect(x, ...)

## S3 method for class 'query'
collect(x, ...)

## S3 method for class 'query_set'
collect(x, ..., wait = TRUE, file = NULL)

## S3 method for class 'computed_query'
collect(x, ..., wait = TRUE, file = NULL)
```

Arguments

x	An object of class data_request, metadata_request or files_request (from galah_call()); or an object of class prequery, query_set or query (from capture(), collapse() or compute())
...	Arguments passed on to other methods
wait	logical; should galah wait for a response? Defaults to FALSE. Only applies for type = "occurrences" or "species".
file	(Optional) file name. If not given, will be set to data with date and time added. The file path (directory) is always given by galah_config()$package$directory.

Details

galah uses an object-based pipeline to convert piped requests into valid queries, and to enact those queries with the specified organisation. Typically, requests open with galah_call() - though request_metadata() and request_files() are also valid - and end with collect(). Under the hood, the sequence of functions is as follows:

capture() → compound() → collapse() → compute() → collect()

collect() is the final step of the galah_call() workflow, and it retrieves the result of a query once it is processed by the server.

Value

In most cases, collect() returns a tibble containing requested data. Where the requested data are not yet ready (i.e. for occurrences when wait is set to FALSE), this function returns an object of class query that can be used to recheck the download at a later time.

See Also

To open a piped query, see galah_call(). For alternative operations on _request objects, see capture(), compound(), collapse() or compute().

---

collect_media *Collect media files*

Description

This function downloads full-sized or thumbnail images and media files to a local directory using information from atlas_media()

Usage

```
collect_media(df, thumbnail = FALSE)
```

Arguments

df	A tibble returned by atlas_media() or a pipe starting with request_data(type = "media").
thumbnail	Default is FALSE. If TRUE will download small thumbnail-sized images, rather than full size images (default).

Value

Invisibly returns a tibble listing the number of files downloaded, grouped by their HTML status codes. Primarily called for the side effect of downloading available image & media files to a user local directory.

Examples

```
## Not run:
# Use `atlas_media()` to return a `tibble` of records that contain media
x <- galah_call() |>
  identify("perameles") |>
  filter(year == 2015) |>
  atlas_media()

# To download media files, add `collect_media()` to the end of a query
galah_config(directory = "media_files")
collect_media(x)

# Since version 2.0, it is possible to run all steps in sequence
# first, get occurrences, making sure to include media fields:
occurrences_df <- request_data() |>
  identify("Regent Honeyeater") |>
  filter(!is.na(images), year == 2011) |>
  select(group = "media") |>
```

```
  collect()

# second, get media metadata
media_info <- request_metadata() |>
  filter(media == occurrences_df) |>
  collect()

# the two steps above + `right_join()` are synonymous with `atlas_media()`
# third, get images
request_files() |>
  filter(media == media_info) |>
  collect(thumbnail = TRUE)
# step three is synonymous with `collect_media()`

## End(Not run)
```

---

compound *Force evaluation of a database query*

Description

`compound()` shows the full set of queries required to properly evaluate the user's request, run prior to `collapse()`.

The number of total queries to send for a single data request is often broader than the single query returned by `collapse()`. If, for example, the user's query includes a call to `identify()`, then a taxonomic query is required to run *before* the 'final' query is attempted. In relation to other functions that manipulate `_request` objects, `compound()` is called within `collapse()`, and itself calls `capture()` internally where required.

Usage

```
compound(x, ...)

## S3 method for class 'data_request'
compound(x, mint_doi = FALSE, ...)

## S3 method for class 'metadata_request'
compound(x, ...)

## S3 method for class 'files_request'
compound(x, ...)

## S3 method for class 'prequery'
compound(x, mint_doi = FALSE, ...)

## S3 method for class 'query'
compound(x, ...)
```

```
## S3 method for class 'query_set'
compound(x, ...)
```

Arguments

x	An object to be compounded. Works for data_request, metadata_request, file_request, query or prequery.
...	Other arguments passed to capture().
mint_doi	Logical: should a DOI be minted for this download? Only applies to type = "occurrences", and only for supported atlases.

Details

galah uses an object-based pipeline to convert piped requests into valid queries, and to enact those queries with the specified organisation. Typically, requests open with galah_call() - though request_metadata() and request_files() are also valid - and end with collect(). Under the hood, the sequence of functions is as follows:

capture() → compound() → collapse() → compute() → collect()

compound() is the second of the galah_call() workflow, and it collates the complete list of queries required to send in order to meet the user's data request, returned by collapse().

Value

An object of class query_set, which is simply a list of all query objects required to properly evaluate the specified request. Objects are listed in the order in which they will be evaluated, meaning the query that the user has actually requested will be placed last.

See Also

To open a piped query, see galah_call(). For alternative operations on _request objects, see capture(), collapse(), compute() or collect().

---

compute.data_request *Compute a query*

Description

Sends a request for information to a server. This is useful for requests that run a server-side process, as it separates the submission of the request from its retrieval.

Within galah, compute() is generally hidden as it is one part of the overall process to complete a data_request, metadata_request or file_request. However, calling compute() at the end of a galah_call() sends a request to be completed server-side (i.e., outside of R), and the result can be returned in R by calling collect() at a later time. This can be preferable to calling atlas_occurrences(), which prevents execution of new code until the server-side process is complete.

Usage

```
## S3 method for class 'data_request'
compute(x, ...)

## S3 method for class 'metadata_request'
compute(x, ...)

## S3 method for class 'files_request'
compute(x, ...)

## S3 method for class 'prequery'
compute(x, ...)

## S3 method for class 'query'
compute(x, ...)

## S3 method for class 'query_set'
compute(x, ...)
```

Arguments

x	An object of class `data_request`, `metadata_request` or `files_request` (i.e. constructed using a pipe) or `query` (i.e. constructed by `collapse()`)
...	Arguments passed on to other methods

Details

`galah` uses an object-based pipeline to convert piped requests into valid queries, and to enact those queries with the specified organisation. Typically, requests open with `galah_call()` - though `request_metadata()` and `request_files()` are also valid - and end with `collect()`. Under the hood, the sequence of functions is as follows:

`capture()` → `compound()` → `collapse()` → `compute()` → `collect()`

`compute()` sends a query to a server, which, once completed, can be retrieved using `collect()`.

Value

An object of class `computed_query`, which is identical to class `query` except for occurrence data, where it also contains information on the status of the request.

See Also

To open a piped query, see `galah_call()`. For alternative operations on `_request` objects, see `capture()`, `compound()`, `collapse()`, `collect()`.

count.data_request *Count the observations in each group*

Description

count() lets you quickly count the unique values of one or more variables. It is evaluated lazily. add_count() is an equivalent that uses mutate() to add a new column with group-wise counts.

Usage

```
## S3 method for class 'data_request'
count(x, ..., wt, sort, name)

## S3 method for class 'data_request'
add_count(x, ..., wt = NULL, sort = FALSE, name = NULL)
```

Arguments

x	An object of class data_request, created using galah_call()
...	currently ignored
wt	currently ignored
sort	currently ignored
name	currently ignored

distinct.data_request *Keep distinct/unique rows*

Description

Keep only unique/distinct rows from a data frame. This is similar to unique.data.frame() but considerably faster. It is evaluated lazily.

Usage

```
## S3 method for class 'data_request'
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See Methods, below, for more details.
...	Variables to use when determining uniqueness. Unlike the dplyr implementation this must be set for the function to do anything, and only a single variable is used.
.keep_all	If TRUE, keep all variables in .data. Defaults to FALSE

Details

This function has several potential uses. In it's default mode, it simply shows the unique values for a supplied field:

```
galah_call() |>
  distinct(basisOfRecord) |>
  collect()

# A tibble: 9 × 1
  basisOfRecord
  <chr>
1 HUMAN_OBSERVATION
2 PRESERVED_SPECIMEN
3 OCCURRENCE
4 MACHINE_OBSERVATION
5 OBSERVATION
6 MATERIAL_SAMPLE
7 LIVING_SPECIMEN
8 FOSSIL_SPECIMEN
9 MATERIAL_CITATION
```

This is the same result as you would get using `show_values()`:

```
search_all(fields, "basisOfRecord") |>
  show_values()
```

Using `distinct()` is somewhat more reliable, however, as it doesn't rely on searching the tibble returned by `show_all(fields)`. It is also more efficient, particularly when caching is turned off. If the goal is to retrieve the *number* of levels of a factor, use:

```
galah_call() |>
  distinct(basisOfRecord) |>
  count() |>
  collect()

# A tibble: 1 × 1
  count
  <int>
1     9
```

When the variable passed to `distinct()` in the above example is `speciesID`, this is identical to calling:

```
atlas_counts(type = "species")
```

You can also pass `group_by()` to find the number of facets per level of a second variable:

```
galah_call() |>
  identify("Perameles") |>
  distinct(speciesID) |>
  group_by(basisOfRecord) |>
  count() |>
  collect()
```

```
# A tibble: 8 × 2
  basisOfRecord       count
  <chr>               <int>
1 Human observation       7
2 Preserved specimen      9
3 Machine observation     2
4 Observation             3
5 Occurrence              3
6 Material Sample         4
7 Fossil specimen         1
8 Living specimen         1
```

By setting `.keep_all = TRUE`, we get more information on each record. Due to limits on the APIs this is not a perfect analogy for running `dplyr::distinct()` on raw occurrences; but it does allow us to generalise `atlas_species()` to use any taxonomic identifier. For example, we might choose to show data by family instead of species:

```
galah_call() |>
  identify("Coleoptera") |>
  distinct(familyID, .keep_all = TRUE) |>
  collect()
```

Using `group_by()` is also valid:

```
galah_call() |>
    filter(year == 2024,
           genus == "Crinia") |>
    group_by(speciesID) |>
    distinct(.keep_all = TRUE) |>
    collapse()
```

In this case, `collect()` and `atlas_species()` are synonymous, with the exception that the latter does not require you to set the `.keep_all` argument to `TRUE`. So you could instead use:

```
galah_call() |>
  identify("Coleoptera") |>
  distinct(familyID) |>
  atlas_species()
```

Examples

```
## Not run:
galah_call() |>
  distinct(basisOfRecord) |>
  count() |>
  collect()

## End(Not run)
```

`filter.data_request` *Keep rows that match a condition*

Description

The `filter()` function is used to subset a data, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions. Unlike 'local' filters that act on a `tibble`, the galah implementations work by amending a query which is then enacted by `collect()` or one of the `atlas_` family of functions (such as `atlas_counts()` or `atlas_occurrences()`).

Usage

```
## S3 method for class 'data_request'
filter(.data, ...)

## S3 method for class 'metadata_request'
filter(.data, ...)

## S3 method for class 'files_request'
filter(.data, ...)

galah_filter(...)
```

Arguments

`.data`	An object of class `data_request`, `metadata_request` or `files_request`, created using `galah_call()` or related functions.
`...`	Expressions that return a logical value, and are defined in terms of the variables in the selected atlas (and checked using `show_all(fields)`. If multiple expressions are included, they are combined with the & operator. Only rows for which all conditions evaluate to `TRUE` are kept.

Details

Syntax

`filter()` uses non-standard evaluation (NSE), and is designed to be as compatible as possible with `dplyr::filter()` syntax. Permissible examples include:

- == (e.g. `year = 2020`) but not = (for consistency with `dplyr`)
- `!=`, e.g. `year != 2020`)
- > or >= (e.g. `year >= 2020`)
- < or <= (e.g. `year <= 2020`)
- `OR` statements (e.g. `year == 2018 | year == 2020`)
- `AND` statements (e.g. `year >= 2000 & year <= 2020`)

Some general tips:

- Separating statements with a comma is equivalent to an `AND` statement; Ergo `filter(year >= 2010 & year < 2020)` is the same as `_filter(year >= 2010, year < 2020)`.
- All statements must include the field name; so `filter(year == 2010 | year == 2021)` works, as does `filter(year == c(2010, 2021))`, but `filter(year == 2010 | 2021)` fails.
- It is possible to use an object to specify required values, e.g. `year_value <- 2010; filter(year > year_value)`.
- `solr` supports range queries on text as well as numbers; so `filter(cl22 >= "Tasmania")` is valid.
- It is possible to filter by 'assertions', which are statements about data validity, such as `filter(assertions != c("INVA`. Valid assertions can be found using `show_all(assertions)`.

Exceptions

When querying occurrences, species, or their respective counts (i.e. all of the above examples), field names are checked internally against `show_all(fields)`. There are some cases where bespoke field names are required, as follows.

When requesting a data download from a DOI, the field `doi` is valid, i.e.:

```
galah_call() |>
  filter(doi = "a-long-doi-string") |>
  collect()
```

For taxonomic metadata, the `taxa` field is valid:

```
request_metadata() |>
  filter(taxa == "Chordata") |>
  unnest()
```

For building taxonomic trees, the `rank` field is valid:

```
request_data() |>
  identify("Chordata") |>
  filter(rank == "class") |>
  atlas_taxonomy()
```

Media queries are more involved, but break two rules: they accept the `media` field, and they accept a tibble on the rhs of the equation. For example, users wishing to break down media queries into their respective API calls should begin with an occurrence query:

```
occurrences <- galah_call() |>
   identify("Litoria peronii) |>
   select(group = c("basic", "media") |>
   collect()
```

They can then use the `media` field to request media metadata:

```
media_metadata <- request_metadata |>
  filter(media == occurrences) |>
  collect()
```

And finally, the metadata tibble can be used to request files:

```
request_files() |>
  filter(media == media_metadata) |>
  collect()
```

Value

A tibble containing filter values.

See Also

`select()`, `group_by()` and `geolocate()` for other ways to amend the information returned by `atlas_()` functions. Use `search_all(fields)` to find fields that you can filter by, and `show_values()` to find what values of those filters are available.

Examples

```
## Not run:
# basic example
galah_call() |>
  filter(year >= 2019,
         basisOfRecord == "HumanObservation") |>
  count() |>
  collect()

# Field names can be parsed from objects using `{{}}` syntax, e.g.
field <- "year"
value <- "2025"
galah_call() |>
  filter({{field}} == value) |>
  count() |>
  collect()

## End(Not run)
```

filter_object_classes *Object classes for* filter() *queries*

Description

In galah, there are several ways to provide filter information. To ensure these are handled and printed correctly, they are assigned classes

Usage

```
as_data_filter(x)

as_predicates_filter(x)

as_metadata_filter(x)

as_files_filter(x)

## S3 method for class 'data_filter'
print(x, ...)

## S3 method for class 'predicates_filter'
print(x, ...)

## S3 method for class 'metadata_filter'
print(x, ...)

## S3 method for class 'files_filter'
print(x, ...)
```

Arguments

x	a list, or object of supported class
...	Additional arguments, currently ignored

galah_call *Start building a request*

Description

To download data from the selected atlas, one must construct a query. This query tells the atlas API what data to download and return, as well as how it should be filtered. Using galah_call() allows you to build a piped query to download data, in the same way that you would wrangle data with dplyr and the tidyverse. It is synonymous with request_data(); to query other data types call request_metadata() or request_files().

Usage

```
galah_call(
  type = c("occurrences", "occurrences-count", "occurrences-doi", "species",
    "species-count")
)

request_data(
  type = c("occurrences", "occurrences-count", "occurrences-doi", "species",
    "species-count")
)

request_metadata(
  type = c("fields", "apis", "assertions", "atlases", "collections", "config",
    "datasets", "licences", "lists", "media", "profiles", "providers", "ranks",
    "reasons", "taxa", "identifiers")
)

request_files(type = "media")
```

Arguments

type	string: what form of data should be returned? Acceptable values are specified by the corresponding request function

Details

galah_call() and any of the request_ functions are used to begin a piped query, which is then actioned using collect(), or optionally one of the atlas_ family of functions.

Having distinct functions for different types of request is useful because it allows galah to separate different types of requests to perform better. For example, filter.data_request()translates filters to solr syntax for the living atlases, or to predicates for GBIF, whereas filter.metadata_request() adds a search term to your metadata query.

Value

Each sub-function returns a different object class:

- request_data() and galah_call() return class "data_request"
- request_metadata() returns class "metadata_request"
- request_files() returns class "files_request"

These objects are list-like and store later dplyr verbs in the order they are provided.

See Also

To amend a request object, use apply_profile(), arrange(), count(), distinct(), filter(), glimpse(), group_by(), identify(), select, slice_head() or unnest(). For operations on _request objects, see capture(), compound(), collapse(), compute() or collect().

Examples

```
## Not run:
# Begin your query with `galah_call()`, then pipe using `%>%` or `|>`

# Get number of records of *Aves* from 2001 to 2004 by year
galah_call() |>
  identify("Aves") |>
  filter(year > 2000 & year < 2005) |>
  group_by(year) |>
  count() |>
  collect()

# Get information for all species in *Cacatuidae* family
galah_call() |>
  identify("Cacatuidae") |>
  distinct("speciesID", .keep_all = TRUE) |>
  collect()

# Download records of genus *Eolophus* from 2001 to 2004
galah_config(email = "your-email@email.com")

galah_call() |>
  identify("Eolophus") |>
  filter(year > 2000 & year < 2005) |>
  collect()

## End(Not run)
```

`galah_config` *View or set package behaviour*

Description

The `galah` package supports queries to a number of different data providers, and once selected, it is desirable that all later queries are sent to that organisation. Rather than supply this information separately in each query, it is more parsimonious to cache it centrally and call it as needed, which is what this function supports.

Beyond choosing an organisation, there are several other use cases for caching. Many GBIF nodes require the user to supply a registered email address, password, and (in some cases) a reason for downloading data, all stored via `galah_config()`. This function also provides a convenient place to control optional package behaviours, such as checking queries to ensure they are valid (`run_checks`), informing you via email when your downloads are ready (`send_email`), or controlling whether galah will provide updates on your query as they are processed (`verbose`).

Usage

```
galah_config(...)
```

Arguments

...	Options can be defined using the form `name = "value"`, or as a (single) named list. See details for accepted fields.

Details

Valid arguments to this function are:

- `atlas` string: Living Atlas to point to, Australia by default. Can be an organisation name, acronym, or region (see `show_all_atlases()` for admissible values)
- `authenticate` logical: Should `galah` use authenticate your queries using JWT tokens? Defaults to `FALSE`. If `TRUE`, user credentials are verified prior to sending a query. This can allow users with special access to download additional information in `galah`.
- `caching` logical: should metadata query results be cached in `options()`? Defaults to `TRUE` for improved stability and speed.
- `directory` string: The directory to use for the disk cache. By default this is a temporary directory, which means that results will only be cached within an R session and cleared automatically when the user exits R. The user may wish to set this to a non-temporary directory for caching across sessions. The directory must exist on the file system.
- `download_reason_id` numeric or string: the "download reason" required. by some ALA services, either as a numeric ID (currently 0–13) or a string (see `show_all(reasons)` for a list of valid ID codes and names). By default this is NA. Some ALA services require a valid download_reason_id code, either specified here or directly to the associated R function.
- `email` string: An email address that has been registered with the chosen atlas. For the ALA, you can register at this address.
- `password` string: A registered password (GBIF only)
- `run_checks` logical: should `galah` run checks for filters and columns. If making lots of requests sequentially, checks can slow down the process and lead to HTTP 500 errors, so should be turned off. Defaults to TRUE.
- `send_email` logical: should you receive an email for each query to `atlas_occurrences()`? Defaults to `FALSE`; but can be useful in some instances, for example for tracking DOIs assigned to specific downloads for later citation.
- `username` string: A registered username (GBIF only)
- `verbose` logical: should `galah` give verbose such as progress bars? Defaults to `FALSE`.

Value

Returns an object with classes `galah_config` and `list`, invisibly if arguments are supplied.

Examples

```
## Not run:
# To download occurrence records, enter your email in `galah_config()`.
# This email should be registered with the atlas in question.
galah_config(email = "your-email@email.com")
```

```
# Turn on caching in your session
galah_config(caching = TRUE)

# Some ALA services require that you add a reason for downloading data.
# Add your selected reason using the option `download_reason_id`
galah_config(download_reason_id = 0)

# To look up all valid reasons to enter, use `show_all(reasons)`
show_all(reasons)

# Make debugging in your session easier by setting `verbose = TRUE`
galah_config(verbose = TRUE)

# Optionally supply arguments via a named list
list(email = "your-email@email.com") |>
  galah_config()

## End(Not run)
```

`geolocate` *Narrow a query to within a specified area*

Description

Restrict results to those from a specified area. Areas can be specified as either polygons or bounding boxes, depending on `type`. Alternatively, users can call the underlying functions directly via `galah_polygon()`, `galah_bbox()` or `galah_radius()`. It is possible to use `sf` syntax by calling `st_crop()`, which is synonymous with `galah_polygon()`.

Use a polygon If calling `galah_geolocate()`, the default `type` is `"polygon"`, which narrows queries to within an area supplied as a `POLYGON` or `MULTIPOLYGON`. Polygons must be specified as either an `sf` object, a 'well-known text' (WKT) string, or a shapefile. Shapefiles must be simple to be accepted by the ALA.

Use a bounding box Alternatively, set `type = "bbox"` to narrow queries to within a bounding box. Bounding boxes can be extracted from a supplied `sf` object or a shapefile. A bounding box can also be supplied as a `bbox` object (via `sf::st_bbox()`) or a `tibble`/`data.frame`.

Use a point radius Alternatively, set `type = "radius"` to narrow queries to within a circular area around a specific point location. Point coordinates can be supplied as latitude/longitude coordinate numbers or as an `sf` object (`sfc_POINT`). Area is supplied as a `radius` in kilometres. Default radius is 10 km.

Usage

```
geolocate(..., type = c("polygon", "bbox", "radius"))

galah_geolocate(..., type = c("polygon", "bbox", "radius"))

galah_polygon(...)
```

```
galah_bbox(...)

galah_radius(...)

## S3 method for class 'data_request'
st_crop(x, y, ...)
```

Arguments

...	For st_crop, additional arguments (currently ignored). Otherwise a single sf object, WKT string or shapefile. Bounding boxes can be supplied as a tibble/data.frame or a bbox
type	string: one of c("polygon", "bbox"). Defaults to "polygon". If type = "polygon", a multipolygon will be built via galah_polygon(). If type = "bbox", a multipolygon will be built via galah_bbox(). The multipolygon is used to narrow a query to the ALA.
x	An object of class data_request, created using galah_call()
y	A valid Well-Known Text string (wkt), a POLYGON or a MULTIPOLYGON

Details

If type = "polygon", WKT strings longer than 10000 characters and sf objects with more than 500 vertices will not be accepted by the ALA. Some polygons may need to be simplified. If type = "bbox", sf objects and shapefiles will be converted to a bounding box to query the ALA. If type = "radius, sfc_POINT objects will be converted to lon/lat coordinate numbers to query the ALA. Default radius is 10 km.

Value

If type = "polygon" or type = "bbox", length-1 string (class character) containing a multipolygon WKT string representing the area provided. If type = "radius", list of lat, long and radius values.

Examples

```
## Not run:
# Search for records within a polygon using a shapefile
location <- sf::st_read("path/to/shapefile.shp")
galah_call() |>
  identify("vulpes") |>
  geolocate(location) |>
  count() |>
  collect()

# Search for records within the bounding box of a shapefile
location <- sf::st_read("path/to/shapefile.shp")
galah_call() |>
  identify("vulpes") |>
  geolocate(location, type = "bbox") |>
```

```
  count() |>
  collect()

# Search for records within a polygon using an `sf` object
location <- "POLYGON((142.3 -29.0,142.7 -29.1,142.7 -29.4,142.3 -29.0))" |>
 sf::st_as_sfc()
galah_call() |>
  identify("reptilia") |>
  galah_polygon(location) |>
  count() |>
  collect()

# Search for records using a Well-known Text string (WKT)
wkt <- "POLYGON((142.3 -29.0,142.7 -29.1,142.7 -29.4,142.3 -29.0))"
galah_call() |>
  identify("vulpes") |>
  st_crop(wkt) |>
  count() |>
  collect()

# Search for records within the bounding box extracted from an `sf` object
location <- "POLYGON((142.3 -29.0,142.7 -29.1,142.7 -29.4,142.3 -29.0))" |>
  sf::st_as_sfc()
galah_call() |>
  identify("vulpes") |>
  galah_geolocate(location, type = "bbox") |>
  count() |>
  collect()

# Search for records using a bounding box of coordinates
b_box <- sf::st_bbox(c(xmin = 143, xmax = 148, ymin = -29, ymax = -28),
                     crs = sf::st_crs("WGS84"))
galah_call() |>
  identify("reptilia") |>
  galah_geolocate(b_box, type = "bbox") |>
  count() |>
  collect()

# Search for records using a bounding box in a `tibble` or `data.frame`
b_box <- tibble::tibble(xmin = 148, ymin = -29, xmax = 143, ymax = -21)
galah_call() |>
  identify("vulpes") |>
  galah_geolocate(b_box, type = "bbox") |>
  count() |>
  collect()

# Search for records within a radius around a point's coordinates
galah_call() |>
  identify("manorina melanocephala") |>
  galah_geolocate(lat = -33.7,
                  lon = 151.3,
                  radius = 5,
                  type = "radius") |>
```

```
  count() |>
  collect()

# Search for records with a radius around an `sf_POINT` object
point <- sf::st_sfc(sf::st_point(c(-33.66741, 151.3174)), crs = 4326)
galah_call() |>
  identify("manorina melanocephala") |>
  galah_geolocate(point,
                  radius = 5,
                  type = "radius") |>
  count() |>
  collect()

## End(Not run)
```

`glimpse.data_request` *Get a glimpse of your data*

Description

`glimpse()` is like a transposed version of `print()`: columns run down the page, and data runs across. This makes it possible to see every column in a data frame. It's a little like `str()` applied to a data frame but it tries to show you as much data as possible. This implementation is specific to `galah` and is evaluated lazily. **[Experimental]**

Usage

```
## S3 method for class 'data_request'
glimpse(x, ...)

## S3 method for class 'occurrences_glimpse'
print(x, ...)
```

Arguments

`x`	An object of class `data_request`
`...`	Other arguments, currently ignored

Details

This implementation of `glimpse()` actually involves changing the API call sent to the server, then returning a novel object class with it's own `print()` method.

Examples

```
## Not run:
galah_call() |>
  filter(year >= 2019,
         basisOfRecord == "HumanObservation") |>
```

```
  select(year, basisOfRecord, species) |>
  glimpse() |>
  collect()

## End(Not run)
```

`group_by.data_request` *Group by one or more variables*

Description

Most data operations are done on groups defined by variables. `group_by()` takes a field name (unquoted) and performs a grouping operation. The default behaviour is to use it in combination with `count()` to give information on number of occurrences per level of that field. Alternatively, you can use it without count to get a download of occurrences grouped by that variable. This is particularly useful when used with a taxonomic `ID` field (`speciesID`, `genusID` etc.) as it allows further information to be appended to the result. This is how `atlas_species()` works, for example. See `select()` for details.

Usage

```
## S3 method for class 'data_request'
group_by(.data, ...)

galah_group_by(...)
```

Arguments

`.data`	An object of class `data_request`
`...`	Zero or more individual column names to include

Value

If any arguments are provided, returns a `data.frame` with columns `name` and `type`, as per `select.data_request()`.

Examples

```
## Not run:
# default usage is for grouping counts
galah_call() |>
  group_by(basisOfRecord) |>
  counts() |>
  collect()

# Alternatively, we can use this with an occurrence search
galah_call() |>
  filter(year == 2024,
         genus = "Crinia") |>
  group_by(speciesID) |>
```

```
 collect()
# note that this example is equivalent to `atlas_species()`;
# but using `group_by()` is more flexible.

## End(Not run)
```

`identify.data_request` *Narrow a query by passing taxonomic identifiers*

Description

When conducting a search or creating a data query, it is common to identify a known taxon or group of taxa to narrow down the records or results returned. `identify()` is used to identify taxa you want returned in a search or a data query. Users to pass scientific names or taxonomic identifiers with pipes to provide data only for the biological group of interest.

It is good to use `search_taxa()` and `search_identifiers()` first to check that the taxa you provide to `galah_identify()` return the correct results.

Usage

```
## S3 method for class 'data_request'
identify(x, ...)

## S3 method for class 'metadata_request'
identify(x, ...)

galah_identify(...)
```

Arguments

`x`	An object of class `data_request` or `metadata_request`.
`...`	One or more scientific names.

Value

A `tibble` containing identified taxa.

See Also

`filter()` or `geolocate()` for other ways to filter a query. You can also use `search_taxa()` to check that supplied names are being matched correctly on the server-side; see taxonomic_searches for a detailed overview.

Examples

```
## Not run:
# Use `galah_identify()` to narrow your queries
galah_call() |>
  identify("Eolophus") |>
  count() |>
  collect()

# If you know a valid taxon identifier, use `filter()` instead.
id <- "https://biodiversity.org.au/afd/taxa/009169a9-a916-40ee-866c-669ae0a21c5c"
galah_call() |>
  filter(lsid == id)  |>
  count() |>
  collect()

## End(Not run)
```

---

print_galah_objects *Print galah objects*

Description

As of version 2.0, galah supports several bespoke object types. Classes `data_request`, `metadata_request` and `files_request` are for starting pipes to download different types of information. These objects are parsed using `collapse()` into a `query` object, which contains one or more URLs necessary to return the requested information. This object is then passed to `compute()` and/or `collect()`. Finally, `galah_config()` creates an object of class `galah_config` which (unsurprisingly) stores configuration information.

Usage

```
## S3 method for class 'data_request'
print(x, ...)

## S3 method for class 'files_request'
print(x, ...)

## S3 method for class 'metadata_request'
print(x, ...)

## S3 method for class 'query'
print(x, ...)

## S3 method for class 'prequery'
print(x, ...)

## S3 method for class 'computed_query'
print(x, ...)
```

```
## S3 method for class 'query_set'
print(x, ...)

## S3 method for class 'galah_config'
print(x, ...)
```

Arguments

x	an object of the appropriate class
...	Arguments to be passed to or from other methods

Value

Print does not return an object; instead it prints a description of the object to the console

Examples

```
## Not run:
# The most common way to start a pipe is with `galah_call()`
# later functions update the `data_request` object
galah_call() |> # same as calling `request_data()`
  filter(year >= 2020) |>
  group_by(year) |>
  count()

# Metadata requests are formatted in a similar way
request_metadata() |>
  filter(field == basisOfRecord) |>
  unnest()

# Queries are converted into a `query_set` by `collapse()`
x <- galah_call() |> # same as calling `request_data()`
  filter(year >= 2020) |>
  count() |>
  collapse()
print(x)

# Each `query_set` contains one or more `query` objects
x[[3]]

## End(Not run)
```

---

read_zip *Read downloaded data from a zip file*

Description

Living atlases supply data downloads as zip files. This function reads these data efficiently, i.e. without unzipping them first, using the `readr` package. Although this function has been part of galah for some time, it was previously internal to `atlas_occurrences()`. It has been exported now to support easy re-importing of downloaded files, without the need to re-run a query.

Usage

```
read_zip(file)
```

Arguments

`file`	(character) A file name. Must be a length-1 character ending in `.zip`.

Examples

```
## Not run:
# set a working directory
galah_config(directory = "data-raw",
             email = "an-email-address@email.com")

# download some data
galah_call() |>
  identify("Heleioporus") |>
  filter(year == 2022) |>
  collect(file = "burrowing_frog_data.zip")

# load data from file
x <- read_zip("./data-raw/burrowing_frog_data.zip")

## End(Not run)
```

`search_all` *Search for record information*

Description

The living atlases store a huge amount of information, above and beyond the occurrence records that are their main output. In `galah`, one way that users can investigate this information is by searching for a specific option or category for the type of information they are interested in. Functions prefixed with `search_` do this, displaying any matches to a search term within the valid options for the information specified by the suffix.

For more information about taxonomic searches using `search_taxa()`, see `?taxonomic_searches`.

`search_all()` is a helper function that can do searches for multiple types of information, acting as a wrapper around many `search_` sub-functions. See `Details` (below) for accepted values.

Usage

```
search_all(type, query, all_fields = FALSE)

search_assertions(query, all_fields = FALSE)

search_apis(query, all_fields = FALSE)

search_atlases(query, all_fields = FALSE)

search_collections(query, all_fields = FALSE)

search_datasets(query, all_fields = FALSE)

search_fields(query, all_fields = FALSE)

search_identifiers(..., all_fields = FALSE)

search_licences(query, all_fields = FALSE)

search_lists(query, all_fields = FALSE)

search_media(query, all_fields = FALSE)

search_profiles(query, all_fields = FALSE)

search_providers(query, all_fields = FALSE)

search_ranks(query, all_fields = FALSE)

search_reasons(query, all_fields = FALSE)

search_taxa(..., all_fields = FALSE)
```

Arguments

type	A string to specify what type of parameters should be searched.
query	A string specifying a search term. Searches are not case-sensitive.
all_fields	**[Experimental]** If TRUE, show_values() also returns all columns available from the API, rather than the 'default' columns traditionally provided via galah.
...	One or more objects accepted by the taxonomic lookup services. See taxonomic_searches for details

Details

There are six categories of information, each with their own specific sub-functions to look-up each type of information. The available types of information for search_all() are:

Category	Type	Description	Sub-fun
configuration	`atlases`	Search for what atlases are available	`search_`
	`apis`	Search for what APIs & functions are available for each atlas	`search_`
	`reasons`	Search for what values are acceptable as 'download reasons' for a specified atlas	`search_`
taxonomy	`taxa`	Search for one or more taxonomic names	`search_`
	`identifiers`	Take a universal identifier and return taxonomic information	`search_`
	`ranks`	Search for valid taxonomic ranks (e.g. Kingdom, Class, Order, etc.)	`search_`
filters	`fields`	Search for fields that are stored in an atlas	`search_`
	`assertions`	Search for results of data quality checks run by each atlas	`search_`
	`licenses`	Search for copyright licences applied to media	`search_`
group filters	`profiles`	Search for what data profiles are available	`search_`
	`lists`	Search for what species lists are available	`search_`
data providers	`providers`	Search for which institutions have provided data	`search_`
	`collections`	Search for the specific collections within those institutions	`search_`
	`datasets`	Search for the data groupings within those collections	`search_`
media	`media`	Search for images or sounds using a vector of IDs	`search_`

Value

An object of class `tbl_df` and `data.frame` (aka a tibble) containing all data that match the search query.

See Also

Use the `show_all()` function and `show_all_()` sub-functions to show available options of information. These functions are used to pass valid arguments to `filter()`, `select()`, and related functions. Taxonomic queries are somewhat more involved; see taxonomic_searches for details.

Examples

```
## Not run:
# Search for fields that include the word "date"
search_all(fields, "date")

# Search for fields that include the word "marine"
search_all(fields, "marine")

# Search using a single taxonomic term
# (see `?search_taxa()` for more information)
search_all(taxa, "Reptilia") # equivalent

# Look up a unique taxon identifier
# (see `?search_identifiers()` for more information)
search_all(identifiers,
           "https://id.biodiversity.org.au/node/apni/2914510")

# Search for species lists that match "endangered"
search_all(lists, "endangered") # equivalent

# Search for a valid taxonomic rank, "subphylum"
```

```
search_all(ranks, "subphylum")

# An alternative is to download the data and then `filter` it. This is
# largely synonymous, and allows greater control over which fields are searched.
request_metadata(type = "fields") |>
 collect() |>
 dplyr::filter(grepl("date", id))

## End(Not run)
```

`select.data_request` *Keep or drop columns using their names*

Description

Select (and optionally rename) variables in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name. Note that unlike calling `select()` on a local tibble, this implementation is only evaluated at the `collapse()` stage, meaning any errors or messages will be triggered at the end of the pipe.

`select()` supports `dplyr` **selection helpers**, including:

- `everything`: Matches all variables. This is treated unusually in `galah`; see `details`.
- `last_col`: Select last variable, possibly with an offset.

Other helpers select variables by matching patterns in their names:

- `starts_with`: Starts with a prefix.
- `ends_with`: Ends with a suffix.
- `contains`: Contains a literal string.
- `matches`: Matches a regular expression.
- `num_range`: Matches a numerical range like x01, x02, x03.

Or from variables stored in a character vector:

- `all_of`: Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of`: Same as `all_of()`, except that no error is thrown for names that don't exist.

Or using a predicate function:

- `where`: Applies a function to all variables and selects those for which the function returns `TRUE`.

Usage

```
## S3 method for class 'data_request'
select(.data, ..., group = NULL)

## S3 method for class 'metadata_request'
select(.data, ...)

galah_select(..., group = NULL)
```

Arguments

.data	An object of class data_request, created using galah_call().
...	Zero or more individual column names to include.
group	string: (optional) name of one or more column groups to include. Valid options are "basic", "event" "taxonomy", "media" and "assertions".

Details

GBIF nodes store content in hundreds of different fields, and users often require thousands or millions of records at a time. To reduce time taken to download data, and limit complexity of the resulting tibble, it is sensible to restrict the fields returned by occurrence queries. The full list of available fields can be viewed with show_all(fields). Note that select() and galah_select() are supported for all atlases that allow downloads, with the exception of GBIF, for which all columns are returned.

Calling the argument group = "basic" returns the following columns:

- recordID
- scientificName
- taxonConceptID
- decimalLatitude
- decimalLongitude
- eventDate
- basisOfRecord
- occurrenceStatus
- dataResourceName

Using group = "event" returns the following columns:

- eventRemarks
- eventTime
- eventID
- eventDate
- samplingEffort
- samplingProtocol

Using group = "media" returns the following columns:

- multimedia
- multimediaLicence
- images
- videos
- sounds

Using group = "taxonomy" returns higher taxonomic information for a given query. It is the only group that is accepted by atlas_species() as well as atlas_occurrences().

Using group = "assertions" returns all quality assertion-related columns. The list of assertions is shown by show_all_assertions().

For atlas_occurrences(), arguments passed to ... should be valid field names, which you can check using show_all(fields). For atlas_species(), it should be one or more of:

- counts to include counts of occurrences per species.
- synonyms to include any synonymous names.
- lists to include authoritative lists that each species is included on.

For metadata queries - as generated using request_metadata() or galah_call() - select() can now be used to return only the requested columns. Unlike data queries, this works by capturing the user's query and applying it user-side, rather than amending the query.

Value

A tibble specifying the name and type of each column to include in the call to atlas_counts() or atlas_occurrences().

See Also

filter(), st_crop() and identify() for other ways to restrict the information returned; show_all(fields) to list available fields.

Examples

```
## Not run:
# Download occurrence records of *Perameles*,
# Only return scientificName and eventDate columns
galah_config(email = "your-email@email.com")
galah_call() |>
  identify("perameles")|>
  select(scientificName, eventDate) |>
  collect()

# Only return the "basic" group of columns and the basisOfRecord column
galah_call() |>
  identify("perameles") |>
  select(basisOfRecord, group = "basic") |>
  collect()
```

```
# When used in a pipe, `galah_select()` and `select()` are synonymous.
# Hence the previous example can be rewritten as:
galah_call() |>
  galah_identify("perameles") |>
  galah_select(basisOfRecord, group = "basic") |>
  collect()

## End(Not run)
```

---

show_all *Show valid record information*

Description

The living atlases store a huge amount of information, above and beyond the occurrence records that are their main output. In galah, one way that users can investigate this information is by showing all the available options or categories for the type of information they are interested in. Functions prefixed with show_all_ do this, displaying all valid options for the information specified by the suffix.

show_all() is a helper function that can display multiple types of information from show_all_ sub-functions.

Usage

```
show_all(..., limit = NULL, all_fields = FALSE)

show_all_apis(limit = NULL, all_fields = FALSE)

show_all_assertions(limit = NULL, all_fields = FALSE)

show_all_atlases(limit = NULL, all_fields = FALSE)

show_all_collections(limit = NULL, all_fields = FALSE)

show_all_config()

show_all_datasets(limit = NULL, all_fields = FALSE)

show_all_fields(limit = NULL, all_fields = FALSE)

show_all_licences(limit = NULL, all_fields = FALSE)

show_all_lists(limit = NULL, all_fields = FALSE)

show_all_profiles(limit = NULL, all_fields = FALSE)
```

```
show_all_providers(limit = NULL, all_fields = FALSE)

show_all_ranks(limit = NULL, all_fields = FALSE)

show_all_reasons(limit = NULL, all_fields = FALSE)
```

Arguments

`...`	String showing what type of information is to be requested. See `Details` (below) for accepted values.
`limit`	Optional number of values to return. Defaults to NULL, i.e. all records
`all_fields`	**[Experimental]** If `TRUE`, `show_values()` also returns all columns available from the API, rather than the 'default' columns traditionally provided via galah.

Details

There are five categories of information, each with their own specific sub-functions to look-up each type of information. The available types of information for `show_all_` are:

Category	Type	Description	Sub-functio
Configuration	`atlases`	Show what atlases are available	`show_all_at`
	`apis`	Show what APIs & functions are available for each atlas	`show_all_ap`
	`config`	Show information necessary for authentication	`show_all_co`
	`reasons`	Show what values are acceptable as 'download reasons' for a specified atlas	`show_all_re`
Data providers	`providers`	Show which institutions have provided data	`show_all_pr`
	`collections`	Show the specific collections within those institutions	`show_all_co`
	`datasets`	Shows all the data groupings within those collections	`show_all_da`
Filters	`assertions`	Show results of data quality checks run by each atlas	`show_all_as`
	`fields`	Show fields that are stored in an atlas	`show_all_fi`
	`licenses`	Show what copyright licenses are applied to media	`show_all_li`
	`profiles`	Show what data profiles are available	`show_all_pr`
Taxonomy	`lists`	Show what species lists are available	`show_all_li`
	`ranks`	Show valid taxonomic ranks (e.g. Kingdom, Class, Order, etc.)	`show_all_ra`

Value

An object of class `tbl_df` and `data.frame` (aka a tibble) containing all data of interest.

References

- Darwin Core terms https://dwc.tdwg.org/terms/

See Also

Use the `search_all()` function and `search_()` sub-functions to search for information. These functions are used to pass valid arguments to `filter()`, `select()`, and related functions.

Examples

```
## Not run:
# See all supported atlases
show_all(atlases)

# Show a list of all available data quality profiles
show_all(profiles)

# Show a listing of all accepted reasons for downloading occurrence data
show_all(reasons)

# Show a listing of all taxonomic ranks
show_all(ranks)

# `show_all()` is synonymous with `request_metadata() |> collect()`
request_metadata(type = "fields") |>
  collect()

# using `all_fields = TRUE` is synonymous with `select(everything())`
request_metadata(type = "fields") |>
  select(everything()) |>
  collect()

## End(Not run)
```

---

show_values *Show or search for values within a specified field*

Description

Users may wish to see the specific values *within* a chosen field, profile or list to narrow queries or understand more about the information of interest. `show_values()` provides users with these values. `search_values()` allows users for search for specific values within a specified field.

Usage

```
show_values(df, all_fields = FALSE)

search_values(df, query)
```

Arguments

`df`	A search result from `search_fields()`, `search_profiles()` or `search_lists()`.
`all_fields`	**[Experimental]** If `TRUE`, `show_values()` also returns all columns available from the API, rather than the 'default' columns traditionally provided via galah. For lists, this will include 'raw' columns; columns included prior to the dataset's ingestion into the ALA, and will often include raw scientific names and vernacular names. For conservation lists like the EPBC list, this also includes columns containing each species' conservation status information.

For other forms of metadata, setting this to TRUE may return more information than you want or need. Default is set to FALSE.

query A string specifying a search term. Not case sensitive.

Details

Each **Field** contains categorical or numeric values. For example:

- The field "year" contains values 2021, 2020, 2019, etc.
- The field "stateProvince" contains values New South Wales, Victoria, Queensland, etc. These are used to narrow queries with filter().

Each **Profile** consists of many individual quality filters. For example, the "ALA" profile consists of values:

- Exclude all records where spatial validity is FALSE
- Exclude all records with a latitude value of zero
- Exclude all records with a longitude value of zero

Each **List** contains a list of species, usually by taxonomic name. For example, the Endangered Plant species list contains values:

- Acacia curranii (Curly-bark Wattle)
- Brachyscome papillosa (Mossgiel Daisy)
- Solanum karsense (Menindee Nightshade)

Value

A tibble of values for a specified field, profile or list.

Examples

```
## Not run:
# Show values in field 'cl22'
search_fields("cl22") |>
  show_values()

# This is synonymous with `request_metadata() |> unnest()`.
# For example, the previous example can be run using:
request_metadata() |>
  filter(field == "cl22") |>
  unnest() |>
  collect()

# Search for any values in field 'cl22' that match 'tas'
search_fields("cl22") |>
  search_values("tas")

# See items within species list "dr19257"
search_lists("dr19257") |>
  show_values()
```

```
## End(Not run)
```

`slice_head.data_request` *Subset rows using their positions*

Description

`slice()` lets you index rows by their (integer) locations. For objects of classes `data_request` or `metadata_request`, only `slice_head()` is currently implemented, and selects the first `n` rows.

If `.data` has been grouped using `group_by()`, the operation will be performed on each group, so that (e.g.) `slice_head(df, n = 5)` will select the first five rows in each group.

Usage

```
## S3 method for class 'data_request'
slice_head(.data, ..., n, prop, by = NULL)

## S3 method for class 'metadata_request'
slice_head(.data, ..., n, prop, by = NULL)
```

Arguments

`.data`	An object of class `data_request`, created using `galah_call()`
`...`	Currently ignored
`n`	The number of rows to be returned. If data are grouped `group_by()`, this operation will be performed on each group.
`prop`	Currently ignored.
`by`	Currently ignored.

Value

An amended `data_request` with a completed `slice` slot.

Examples

```
## Not run:
# Limit number of rows returned to 3.
# In this case, our query returns the top 3 years with most records.
galah_call() |>
  identify("perameles") |>
  filter(year > 2010) |>
  group_by(year) |>
  count() |>
  slice_head(n = 3) |>
  collect()

## End(Not run)
```

taxonomic_searches *Look up taxon information*

Description

search_taxa() allows users to look up taxonomic names, and ensure they are being matched correctly, before downloading data from the specified organisation.

By default, names are supplied as strings; but users can also specify taxonomic levels in a search using a data.frame or tibble. This is useful when the taxonomic *level* of the name in question needs to be specified, in addition to it's identity. For example, a common method is to use the scientificName column to list a Latinized binomial, but it is also possible to list these separately under genus and specificEpithet (respectively). A more common use-case is to distinguish between homonyms by listing higher taxonomic units, by supplying columns like kingdom, phylum or class.

search_identifiers() allows users to look up matching taxonomic names using their unique taxonConceptID. In the ALA, all records are associated with an identifier that uniquely identifies the taxon to which that record belongs. Once those identifiers are known, this function allows you to use them to look up further information on the taxon in question. Effectively this is the inverse function to search_taxa(), which takes names and provides identifiers.

Note that when taxonomic look-up is required within a pipe, the equivalent to search_taxa() is identify() (or galah_identify()). The equivalent to search_identifiers() is to use filter() to filter by taxonConceptId.

Details

search_taxa() returns the taxonomic match of a supplied text string, along with the following information:

- search_term: The search term used by the user. When multiple search terms are provided in a tibble, these are displayed in this column, concatenated using _.
- scientific_name: The taxonomic name matched to the provided search term, to the lowest identified taxonomic rank.
- taxon_concept_id: The unique taxonomic identifier.
- rank: The taxonomic rank of the returned result.
- match_type: (ALA only) The method of name matching used by the name matching service. More information can be found on the name matching github repository.
- issues: Any errors returned by the name matching service (e.g. homonym, indeterminate species match). More information can be found on the name matching github repository.
- taxonomic names (e.g. kingdom, phylum, class, order, family, genus)

When querying using request_metadata(), you have the option to pass select() within the query. The easiest way to do this is select(everything()), but for completeness, the following additional fields are available:

- success: Logical indicating success or failure of the search

- `scientific_name_authorship`: Author and year for the name in question
- `name_type`: Usually `"SCIENTIFIC"`
- `lft` and `rgt`: Numeric indices for taxonomic lookups
- `species_group` and `species_subgroup`: List-columns giving group names
- `_id` fields for `rank` and any taxonomic rank fields (`kingdom_id`, `phylum_id` etc.)

See Also

`search_all()` for how to get names if taxonomic identifiers are already known. `filter()`, `select()`, `identify()` and `geolocate()` for ways to restrict the information returned by `atlas_()` functions.

Examples

```
## Not run:
# Search using a single string.
# Note that `search_taxa()` is not case sensitive
search_taxa("Reptilia")

# Search using multiple strings.
# `search_taxa()` will return one row per taxon
search_taxa("reptilia", "mammalia")

# Search using more detailed strings with authorship information
search_taxa("Acanthocladium F.Muell")

# Specify taxonomic levels in a tibble using "specificEpithet"
search_taxa(tibble::tibble(
  class = "aves",
  family = "pardalotidae",
  genus = "pardalotus",
  specificEpithet = "punctatus"))

# Specify taxonomic levels in a tibble using "scientificName"
search_taxa(tibble::tibble(
  family = c("pardalotidae", "maluridae"),
  scientificName = c("Pardalotus striatus striatus", "malurus cyaneus")))

# Use OOP for the same effect
# `identify()` tells the code that we want to search for _taxonomic_ metadata.
request_metadata() |>
  identify("crinia") |>
  collect()

# This approach has the advantage that we can call `select()`
request_metadata() |>
  identify("crinia") |>
  select(everything()) |>
  collect()

# Look up a unique taxon identifier
search_identifiers(query = "https://id.biodiversity.org.au/node/apni/2914510")
```

```
# OOP process for identifiers uses `filter()`, not `identify()`
# In these cases the `field` argument is used to specify `type`
request_metadata() |>
  filter(identifier = "https://id.biodiversity.org.au/node/apni/2914510") |>
  select(everything()) |>
  collect()

## End(Not run)
```

`unnest` *Unnest a query*

Description

This syntax is borrowed from `tidyr`, and is conceptually used in the same way here, but in galah unnest amends the query to unnest information server-side, rather than on your machine. It powers all of the `show_values()` functions in galah.

Usage

```
unnest(.query)
```

Arguments

`.query`	An object of class `metadata_request`

Details

Re-implementing existing functions has the consequence of supporting consistent syntax with tidyverse, at the cost of potentially introducing conflicts. This can be avoided by using the `::` operator where required.

Value

An object of class `metadata_request`

Examples

```
## Not run:
# Return values of field `basisOfRecord`
request_metadata() |>
  unnest() |>
  filter(field == basisOfRecord) |>
  collect()

# Using `galah::unnest()` in this way is equivalent to:
show_all(fields, "basisOfRecord") |>
  show_values()
```

```
# to add information to a species list:
request_metadata() |>
  filter(list == "dr650") |>
  select(everything()) |>
  unnest() |>
  collect()

## End(Not run)
```

Index