

Package: **datom** (via **r-universe**)

October 3, 2026

Title A Unified Framework for Versioned, Traceable Tabular Data

Version 0.2.0

Description Provides versioned storage for tabular data without a database or a server. Each table is written as an immutable, content-addressed version -- identical content is detected and stored only once -- while its version history and metadata are kept as code in a 'git' repository and the data itself in a local filesystem or cloud object storage ('S3'). Any past version can be read back exactly by its identifier, and each table records the sources it was derived from, so a project carries full data lineage. A lightweight reader role retrieves current or historical data from storage alone, without 'git' or write access, giving downstream analyses and pipelines a single versioned source of truth. It targets analytical and scientific data management, such as preparing clinical study datasets, and is designed as a foundation for higher-level governance tooling.

License MIT + file LICENSE

URL <https://github.com/amashadihossein/datom>,
<https://amashadihossein.github.io/datom/>

BugReports <https://github.com/amashadihossein/datom/issues>

Depends R (>= 4.1.0)

Imports arrow, cli, digest, fs, glue, httr2, jsonlite, paws.storage,
purrr, rlang, utils, yaml

Suggests covr, git2r, knitr, mockery, rio, rmarkdown, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

VignetteBuilder knitr

Config/pak/sysreqs cmake make libuv1-dev libxml2-dev libssl-dev

Repository <https://amashadihossein.r-universe.dev>

Date/Publication 2026-10-03 18:01:36 UTC

RemoteUrl <https://github.com/amashadihossein/datom>

RemoteRef HEAD

RemoteSha 6da9a64b8d8f32e2720cc4629100781de7ea0d0b

Contents

datom-package	3
datom_add_member	4
datom_assemble_set	7
datom_check_hashable	9
datom_clone	10
datom_example_cutoffs	11
datom_example_data	12
datom_fetch_member	13
datom_get_conn	15
datom_get_lineage	16
datom_get_parents	18
datom_get_set	19
datom_history	21
datom_init_repo	23
datom_lineage_union	25
datom_list	26
datom_list_members	27
datom_member	28
datom_parent	30
datom_projects	32
datom_pull	33
datom_read	34
datom_remove_members	35
datom_repo_attach_governance	38
datom_repo_commit	39
datom_repo_delete	41
datom_repo_push	42
datom_repo_set_data_store	44
datom_status	45
datom_storage_copy	46
datom_storage_delete_prefix	48
datom_storage_list	49
datom_storage_read_json	50
datom_storage_verify	51
datom_store	53
datom_store_local	55
datom_store_s3	55

datom_store_s3_creds	56
datom_structure_members	57
datom_summary	59
datom_sync	60
datom_sync_manifest	62
datom_update_members	64
datom_validate	67
datom_write	69
datom_write_set	71
is_datom_store	75
is_datom_store_local	76
is_datom_store_s3	77
is_datom_store_s3_creds	78
is_valid_datom_repo	78
print.datom_conn	79
print.datom_link	80
print.datom_set	81
print.datom_store	81
print.datom_store_local	82
print.datom_store_s3	83
print.datom_store_s3_creds	83
print.datom_summary	84

Index	86
--------------	-----------

datom-package	<i>datom: A Unified Framework for Versioned, Traceable Tabular Data</i>
---------------	---

Description

Provides versioned storage for tabular data without a database or a server. Each table is written as an immutable, content-addressed version – identical content is detected and stored only once – while its version history and metadata are kept as code in a 'git' repository and the data itself in a local filesystem or cloud object storage ('S3'). Any past version can be read back exactly by its identifier, and each table records the sources it was derived from, so a project carries full data lineage. A lightweight reader role retrieves current or historical data from storage alone, without 'git' or write access, giving downstream analyses and pipelines a single versioned source of truth. It targets analytical and scientific data management, such as preparing clinical study datasets, and is designed as a foundation for higher-level governance tooling.

Key terms

- **Project:** one body of data you manage together, such as one clinical study. It has a name, a GitHub repository that records every change, and a storage location that holds the data itself; the data never goes into git. Created once with `datom_init_repo()`.
- **Store:** tells datom where a project's data lives (a local folder or an S3 bucket) and, if you will write, your GitHub token. Built with `datom_store()`.

- **Developer and reader:** the two roles. A developer has a GitHub token and a local copy of the project, and can write. A reader needs only access to the storage – no token, no git – and can only read. datum picks the role from whether the store carries a token.
- **Connection** (conn): a pointer to one project, returned by `datum_get_conn()`. It records which project, where its data is kept, and your role, and it is the first argument to almost every other function.
- **Table:** a data frame saved in datum under a name, such as dm. Saved with `datum_write()` or `datum_sync()`, read with `datum_read()`.
- **Version:** a long identifier for one exact saved state of a table or a set. Old versions are never overwritten, and a save that changes nothing makes no new version. `datum_history()` lists them; pass one as version = to read it back.
- **Sync manifest:** the preview `datum_sync_manifest()` returns – one row per file in input_files/, each marked new, changed, unchanged, or in a format datum cannot read – which you then pass to `datum_sync()`.
- **Set:** a named, versioned list of exact versions of tables (or other sets), so a whole collection can be cited with one version string. It holds no data.
- **Member:** one entry in a set – one table or set, pinned at one version, with optional labels such as type = "input".
- **Parent, source and lineage:** a parent is a table another table was made from, named with `datum_parent()` before writing. A source is an original imported table at the start of the chain. Lineage is the record of both, read with `datum_get_lineage()`.

Author(s)

Maintainer: Afshin Mashadi-Hosseini <amashadihossein@gmail.com> [copyright holder]

See Also

Useful links:

- <https://github.com/amashadihossein/datum>
- <https://amashadihossein.github.io/datum/>
- Report bugs at <https://github.com/amashadihossein/datum/issues>

datum_add_member

Add One Member to a Set

Description

Declares one member and appends it to a set – an empty one from `datum_assemble_set()` or one read back with `datum_get_set()` – validating it immediately: the artifact must exist at the version given, and its labels must be well formed. The member record is built through the same path `datum_member()` uses, so a set assembled this way is byte-identical to the same set passed as a list.

Usage

```
datum_add_member(x, member, version = NULL, tags = NULL, conn = NULL)
```

Arguments

x	A datum_set, from datum_assemble_set() or datum_get_set() .
member	The member to add: an artifact name, a member record, or a link.
version	The version to pin, when member is a name. Required there; refused beside a record or a link.
tags	Optional named list of text labels for this member, when member is a name. Refused beside a record or a link, which carry their own.
conn	A datum_conn from datum_get_conn() for the project a name is looked up in. Required for a name; not used for a record or a link.

Value

The set, one member longer, with its version and data_sha emptied and the addition appended to its datum_edits attribute – or unchanged, when the member was already in it with the same labels.

Naming a member

member accepts the three shapes a caller holds, and the second and third are not merely convenient:

What you pass	What it means
a name	look this artifact up through conn
a member record	use it as given – from datum_member() , or from a set read back
a link	use the member it points at – x\$members[[i]]\$fetch, or a leaf of datum_structure_members()

A name is looked up in one project's storage: the project conn is for. A set holds no connection, so a name always needs conn, whichever project it is in. A record or a link carries its own resolved pointer and needs no connection at all:

```
datum_assemble_set(conn_a) |>
  datum_add_member("dm", v1, conn = conn_a) |>      # this project, by name
  datum_add_member("ae", v2, conn = conn_b) |>      # another project, by name
  datum_add_member(datum_member(conn_b, "vs", v3)) # another project, as a record
```

A link is how a consumer cites what they used. Someone holding only a projection of a set – a leaf of [datum_structure_members\(\)](#) – can add exactly the version they read to a new set, without reconstructing the pointer.

version and tags describe a member given by **name**. Beside a record or a link they are refused rather than ignored, because a record already carries its own version and labels and a second set of them could only disagree.

Why the version is required

A member pins one exact version, and there is no way to ask for "whatever is current". Inferring current would make a build script produce a **different set** on each run from byte-identical source. Pinning is what makes the artifact immutable; requiring the pin is what makes the code reproducible. List versions with `datom_history()`.

Every add is an edit

Adding a member behaves like `datom_update_members()` and `datom_remove_members()`, whether the set was just assembled or read back:

- **Nothing is written.** The set is stored only when you pass the result to `datom_write_set()`, and the call says so.
- version **and** data_sha **are emptied**, because they described the payload the set was read as. A set never written has neither.
- **The addition is recorded**, so the write's default commit message says add 1 member beside any repoints or removals made on the same object. A freshly assembled set's first write therefore names every member it adds.
- **The new member gets a \$fetch link**, like every member of a set read back.

Adding the same member twice

The two cases differ, and they differ the same way they differ at the write:

- **The same version with the same labels** is skipped, with a note. The write drops an exact repeat anyway, so refusing here would make this verb stricter than the equivalent list – `Reduce(datom_add_member, records, init = x)` over a generated list that happens to repeat would fail where it works today.
- **The same version with different labels** aborts. One version of one artifact is one member holding one set of labels, and merging or choosing between two sets would guess. The write refuses this too; here it names the line that introduced it.

So the member count a set reports is the count the write will produce. Two different **versions** of one artifact are two members, and both are kept.

See Also

`datom_assemble_set()` to start a set, `datom_get_set()` to read one back, `datom_member()` to build a record on another connection, `datom_update_members()` and `datom_remove_members()` for the other edits.

Examples

```
# Adding by name needs a live connection, so the runnable example lives on
# datom_assemble_set(), which shows the whole pipe.
print(names(formals(datom_add_member)))
```

datom_assemble_set	<i>Start Assembling a Set</i>
--------------------	-------------------------------

Description

Returns an empty set, to be filled in with [datom_add_member\(\)](#) and written with [datom_write_set\(\)](#):

Usage

```
datom_assemble_set(conn, name = NULL, tags = NULL)
```

Arguments

conn	A datom_conn from datom_get_conn() for the product repo. Its project name is recorded as the set's project; nothing else is kept.
name	The set's name. NULL (the default) leaves it to the write, which takes the name the repo declares under set: in <code>.datom/project.yaml</code> – the usual case, since one repo holds one set.
tags	Optional named list of set-level text labels, e.g. a description.

Details

```
datom_assemble_set(conn, tags = list(description = "ADaM datasets")) |>
  datom_add_member("adsl", v_adsl, tags = list(type = "output"),
    conn = conn) |>
  datom_add_member("dm", v_dm, tags = list(type = "input"),
    conn = conn_src) |>
  datom_write_set(conn = conn)
```

The equivalent single call – a `list()` of [datom_member\(\)](#) results passed to [datom_write_set\(\)](#) – remains fully supported and is the better fit for a build script. What this path adds is **where an error surfaces**: a malformed member aborts on the line that declared it and names that member, instead of aborting once the whole list has been assembled and indexed.

Value

A [datom_set](#) with no version and no members.

One kind of set

What comes back is a [datom_set](#), the same kind of object [datom_get_set\(\)](#) returns, only with no version yet. So every verb that takes a set takes this one: [datom_list_members\(\)](#), [datom_update_members\(\)](#), [datom_write_set\(\)](#) and the rest.

It holds no connection. A connection may carry a credential, so it is passed on each call that needs one rather than kept in a value that can be printed or saved: `conn` = on [datom_add_member\(\)](#) for a member given by name, and `conn` on [datom_write_set\(\)](#). The connection given here is used only to record which project the set belongs to; the write checks that it is the project it is written into.

Set-level tags

Supplied here rather than by a third verb, because they are facts about the collection rather than about any member. Editing them later is plain R – `x$tags$description <- "..."` – and the same grammar applies as to a member's tags: text only, one label or several.

See Also

`datom_add_member()` to add one member, `datom_write_set()` to write the result, `datom_member()` for the single-call form.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datom_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  datom_write(conn, data = datom_example_data("lb"), name = "lb")

  v_dm <- datom_history(conn, "dm")$version[1]
  v_lb <- datom_history(conn, "lb")$version[1]

  x <- datom_assemble_set(
    conn,
    tags = list(description = "Example product for STUDY-001")
  ) |>
  datom_add_member("dm", v_dm, tags = list(type = "input"), conn = conn) |>
  datom_add_member("lb", v_lb, tags = list(type = "output"), conn = conn)

  print(datom_list_members(x))
  x |> datom_write_set(conn = conn)

  unlink(tmp, recursive = TRUE)
}
```

 datom_check_hashable *Check Whether a Table Can Be Hashed by datom*

Description

Pre-flight check for the datom table contract. Reports, per column, whether `datom_write()` can hash it and – when it cannot – exactly what to do about it. Run this before a write to fix a table in one pass instead of discovering offenders one error at a time.

Usage

```
datom_check_hashable(data)
```

Arguments

`data` A data frame to check.

Details

`datom` identifies a table version by a canonical hash of its contents (`data_sha`), which requires every column to be a supported type: logical, integer, double, character, factor, Date, POSIXct, difftime/hms, `data.table::ITime/IDate`, `bit64::integer64`, or a labelled vector over one of those. List columns (including nested data frames, blobs, and POSIXlt), complex, raw, sf geometry, units, and zoo/chron columns are refused with specific advice.

The advice printed here is the same single-source recourse text `datom_write()` would abort with, so the two can never disagree.

Value

Invisibly, a data frame with one row per column of data and columns:

`column` Column name.

`class` Collapsed class string, or `typeof()` when unclassed.

`status` "ok" or "unsupported".

`recourse` NA when ok, otherwise how to make the column hashable.

See Also

[datom_write\(\)](#)

Examples

```
# A clean table: every column is a supported type
clean <- data.frame(
  id = 1:3,
  score = c(1.5, 2.5, 3.5),
  label = c("a", "b", "c"),
```

```

    grp = factor(c("x", "y", "x")),
    day = as.Date(c("2026-01-01", "2026-01-02", "2026-01-03"))
  )
  datom_check_hashable(clean)

  # An offending table: a list column and a complex column
  messy <- data.frame(id = 1:2)
  messy$notes <- list(c("a", "b"), "c")
  messy$z <- c(1 + 2i, 3 + 4i)
  report <- datom_check_hashable(messy)
  report[report$status == "unsupported", c("column", "recourse")]

```

 datom_clone

Clone a datom Repository

Description

Clones a remote datom repository and returns a connection. This is the recommended way for teammates to join an existing datom project – it wraps `git2r::clone()` and immediately returns a ready-to-use `datom_conn`.

Usage

```
datom_clone(path, store, ...)
```

Arguments

<code>path</code>	Local path to clone into.
<code>store</code>	A <code>datom_store</code> object (from <code>datom_store()</code>). Must have <code>data_repo_url</code> set and role "developer" (i.e., <code>github_pat</code> provided).
<code>...</code>	Additional arguments passed to <code>git2r::clone()</code> .

Details

When `store$gov_repo_url` is set the governance repo is also cloned (or verified if it already exists locally). An existing clone with uncommitted changes causes an error to avoid surprising state.

Value

A `datom_conn` object (developer role).

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)

  # A teammate joins the project from the remote alone.
  conn <- datom_clone(path = file.path(tmp, "teammate"), store = store)
  print(datom_list(conn))

  unlink(tmp, recursive = TRUE)
}
```

datom_example_cutoffs *Monthly Cutoff Dates for Example Study*

Description

Returns a named vector of monthly cutoff dates for STUDY-001, useful for simulating EDC data evolution in examples.

Usage

```
datom_example_cutoffs()
```

Value

Named character vector with entries month_1 through month_6.

Examples

```
datom_example_cutoffs()
# month_1 month_2 month_3 month_4 month_5 month_6
# "2026-01-28" "2026-02-28" ...
```

datom_example_data	<i>Load Example Clinical Trial Data</i>
--------------------	---

Description

Returns one of five small, made-up tables from a simulated clinical trial of 48 subjects: demographics, exposure (dosing), lab results, adverse events or vital signs. Set `cutoff_date` to get the data as it stood on that date, which mimics a new data delivery each month; [datom_example_cutoffs\(\)](#) lists the dates the examples use.

Usage

```
datom_example_data(
  domain = c("dm", "ex", "lb", "ae", "vs"),
  cutoff_date = NULL
)
```

Arguments

<code>domain</code>	One of "dm" (demographics, 48 rows), "ex" (exposure, 48 rows), "lb" (labs, 720 rows: 3 visits x 5 tests per subject), "ae" (adverse events, ~80 rows), or "vs" (vital signs, 432 rows: 3 visits x 3 tests per subject, taken on the same dates as the labs).
<code>cutoff_date</code>	Optional date string ("YYYY-MM-DD") to filter rows whose primary date column is on or before this date, simulating a point-in-time EDC extract. The date column used per domain: RFSTDTC (dm), EXSTDTC (ex), LBDTC (lb), AESTDTC (ae), VSDTC (vs).

Details

The data simulates STUDY-001, a Phase II study enrolling over six months; table and column names loosely follow SDTM.

Value

A data frame.

Examples

```
# Full demographics
dm <- datom_example_data("dm")

# Month-3 snapshot (subjects enrolled by 2026-03-28)
dm_m3 <- datom_example_data("dm", cutoff_date = "2026-03-28")

# Labs collected through Month 3
lb_m3 <- datom_example_data("lb", cutoff_date = "2026-03-28")
```

datom_fetch_member	<i>Get the Data Behind One Member of a Set</i>
--------------------	--

Description

Returns what one [member](#) of a set points at, at the exact version the set records: a data frame for a table, another set for a set. Name the member, as in `datom_fetch_member(conn, x, "dm")`, and pass a connection to that member's own project.

Usage

```
datom_fetch_member(conn, x, member, tags = NULL, version = NULL)
```

Arguments

conn	A <code>datom_conn</code> from datom_get_conn() , scoped to the member's project.
x	A <code>datom_set</code> from datom_get_set() .
member	The member to fetch: its name, a member record, or a link.
tags	Optional named list of labels narrowing an ambiguous name, e.g. <code>list(release = "baseline")</code> . A member matches when it carries every label listed.
version	Optional version, or a prefix of one, narrowing an ambiguous name.

Details

`x$members[[i]]$fetch(conn)` does the same thing.

This is kind dispatch at the **member** level, which is the only level it belongs at. Iterating members, a caller cannot know each one's kind in advance; at the top level they named one artifact they chose, which is why [datom_read\(\)](#) and [datom_get_set\(\)](#) stay separate verbs.

Value

Whatever the member points at: a data frame for a table member, a `datom_set` for a set member.

Naming a member

`member` accepts the three shapes a caller actually holds, so a console call and a loop use one verb:

What you pass	Where it comes from
a name	you read the set and know what you want
a member record	<code>datom_member()</code> , or <code>x\$members[[i]]</code>
a link	<code>x\$members[[i]]\$fetch</code> , or a leaf of <code>datom_structure_members()</code>

A name is not a key. The same artifact at two versions is a legal pair of members – a current table beside a locked baseline, say – so an ambiguous name aborts and lists the candidates rather than answering with the first. Narrow with `tags`, which is the navigation axis, or pin one exactly with `version`. Both narrow a name only; supplied beside a record or a link they are refused rather than quietly ignored.

Which connection to pass

The one for the **member's** project. Access in datom is per project and not conjunctive: reading a set needs the set's project only, and resolving a member is a separate, deliberate step. Same-project members resolve through the connection you already have.

A member's project is **not** checked against the connection's before the fetch, and that is deliberate: a connection's project name is a label the caller supplied and nothing compares it against the repo, so a mismatch is ordinary rather than wrong. When a fetch fails and the two names differ, the error says which project the member's own writer recorded, so the ordinary cause is named instead of presenting as a missing object.

See Also

[datom_list_members\(\)](#) to see every member and its labels, [datom_structure_members\(\)](#) for a navigable view.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datom_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  datom_write(conn, data = datom_example_data("lb"), name = "lb")

  members <- list(
    datom_member(conn, "dm", datom_history(conn, "dm")$version[1],
      tags = list(type = "input")),
    datom_member(conn, "lb", datom_history(conn, "lb")$version[1],
      tags = list(type = "output", domain = c("safety", "labs")))
  )
  datom_write_set(conn, members)

  x <- datom_get_set(conn, "example_product")

  # By name, and the same fetch by the link the read already put on it.
```

```
print(head(datom_fetch_member(conn, x, "dm")))
print(head(datom_fetch_member(conn, x, x$members[[1]]$fetch)))

unlink(tmp, recursive = TRUE)
}
```

datom_get_conn

Get a Pointer to a datom Project

Description

Returns a pointer to the project, called a **connection** (conn): a record of which project you are working on, where its data is kept, and whether you can write. Almost every other datom function takes it as its first argument. Nothing stays open; it only checks once that the storage (and, for a developer, the GitHub repository) can be reached. Developers pass path (their local copy) and store; readers, who have no local copy, pass store and project_name.

Usage

```
datom_get_conn(path = NULL, store = NULL, project_name = NULL, endpoint = NULL)
```

Arguments

path	Path to datom repository. If provided, reads config from .datom/project.yaml.
store	A datom_store object. Required for all connections. The data component provides bucket, prefix, region, and credentials.
project_name	Project name. Required for readers (no local repo). Ignored when path is provided (read from yaml).
endpoint	Optional S3 endpoint URL (e.g., for S3 access points). NULL for default.

Details

Developer (local repo + store): provide path and store. Reads project identity from .datom/project.yaml; uses store for credentials and S3 config. Cross-checks bucket/prefix between yaml and store.

Reader (no local repo): provide store and project_name. Store provides everything.

Value

A datom_conn object.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)

  # Developer: local repo plus store.
  conn <- datom_get_conn(path = file.path(tmp, "repo"), store = store)
  print(conn)

  # Reader: store plus project name, no local repo.
  reader_store <- datom_store(data = datom_store_local(file.path(tmp, "storage")))
  print(datom_get_conn(store = reader_store, project_name = "example_project"))

  unlink(tmp, recursive = TRUE)
}
```

datom_get_lineage

Show a Table's Original Sources or Direct Inputs

Description

Answers "where did this table come from?" from the table's own record. By default (depth = "source") it lists the original imported tables at the start of the chain, skipping the tables in between; depth = "parents" lists only its direct inputs, one step back. Works for any version, and with reader connections.

Usage

```
datom_get_lineage(conn, name, version = NULL, depth = c("source", "parents"))
```

Arguments

conn	A <code>datom_conn</code> object from <code>datom_get_conn()</code> .
name	Table name.
version	Optional <code>metadata_sha</code> (datom version). If <code>NULL</code> , reads current metadata. If provided, fetches the versioned metadata snapshot.
depth	One of "source" (default) or "parents".

Details

It needs access to this table's project only, not to the projects its sources live in.

The two fields answer different questions:

- "source": "what raw datasets does this table ultimately depend on?" (audit, regulatory disclosure, reproducibility scope). Derived at write time as the deduplicated union of the parents' source_lineage fields.
- "parents": "what did this table come from one step back?" (debugging, diff, replay). Equivalent to `datom_get_parents()`.

Value

For depth = "source": the table's recorded source_lineage – a list of source-table descriptors (each with project, table, version_sha), or NULL if the field is absent. For depth = "parents": list of parent entries (each with source, table, version, data_sha), or NULL if no lineage is recorded.

See Also

`datom_get_parents()` for a direct shorthand for the "parents" depth.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  print(datom_get_lineage(conn, "dm", depth = "parents"))
  print(datom_get_lineage(conn, "dm", depth = "source"))

  unlink(tmp, recursive = TRUE)
}
```

datom_get_parents	<i>Get Parent Lineage for a Table</i>
-------------------	---------------------------------------

Description

Reads the parents field from a table's metadata. Returns the lineage entries recorded at write time by [datom_write\(\)](#). For imported tables or derived tables with no recorded lineage, returns NULL.

Usage

```
datom_get_parents(conn, name, version = NULL)
```

Arguments

conn	A datom_conn object from datom_get_conn() .
name	Table name.
version	Optional metadata_sha (datom version). If NULL, reads current metadata. If provided, fetches the versioned metadata snapshot from S3.

Value

List of parent entries (each with source, table, version, data_sha), or NULL if no lineage is recorded. The data_sha field is the parent's authoritative data SHA recorded via [datom_parent\(\)](#), and together with source and version is sufficient to select the parent's project connection and its pinned version.

See Also

[datom_get_lineage\(\)](#) for a unified interface that also exposes the transitive source_lineage field via depth = "source".

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
```

```

conn <- datom_get_conn(file.path(tmp, "repo"), store)

dm <- datom_example_data("dm")
datom_write(conn, data = dm, name = "dm")
datom_write(
  conn,
  data    = dm[dm$SEX == "F", ],
  name    = "dm_female",
  parents = list(datom_parent(conn, "dm", datom_history(conn, "dm")$version[1]))
)
print(datom_get_parents(conn, "dm_female"))

unlink(tmp, recursive = TRUE)
}

```

<i>datom_get_set</i>	<i>Read a datom Set</i>
----------------------	-------------------------

Description

Returns a [set](#)'s members and labels, at its current version or a past one. It reads no table data: to get the data behind a member, use [datom_fetch_member\(\)](#). Works with reader connections.

Usage

```
datom_get_set(conn, name, version = NULL)
```

Arguments

<code>conn</code>	A <code>datom_conn</code> object from datom_get_conn() , scoped to the set's project. A storage-only connection with no git clone is enough.
<code>name</code>	The set's name.
<code>version</code>	Optional version (metadata_sha, or a prefix of one). NULL reads the current version.

Details

Reading a set requires access to the set's own project only. A member is a pointer, and resolving it is a separate, deliberate step – so a 50-member product is readable by someone entitled to none of its members.

Value

A `datom_set`: a list of name, project, version (possibly NULL), data_sha, tags and members.

What comes back

References and labels, and no data at all – which is why the verb is `get` rather than `read`.

A `datom_set`: `name`, `project`, `version`, `data_sha`, `tags` and `members`. The four identifying facts are there so that a caller who passed `version = NULL` can still say which version they got, because a set exists to be cited. `version` is the version **recorded** in the history, so an 8-character prefix goes in and the full version comes back.

`members` is a flat, **unnamed** list in payload order. Not name-keyed, and the reason is not style: the same artifact at two different versions is a legal pair of members, two projects may both hold a `dm`, and R's `$` partial-matches on lists – so a name-keyed list would answer plausibly and wrongly. The unique key is the full id.

One of the four identifying facts has a limit worth knowing before you cite it:

- `version` **can be** `NULL`. It is the version *recorded* in `version_history.json` for the state `metadata.json` describes, and a truncated or partly-synced history records no such entry. A manufactured version would be a wrong statement rather than a missing one, so the field is left empty and `datom_validate()` owns the inconsistency. A version-pinned read always reports one, since the entry is what it resolved through.

`project` is the name the set's **own metadata** records – the declaration of the repo that wrote it, not the name on your connection. It falls back to the connection's name only for a set written by a `datom` that predates the field, which no released build ever was. Each **member** carries its own recorded `id$project` for the same reason, resolved through a slightly longer route because that value is durable and hashed rather than displayed.

Resolving a member

Each member is `id` (`project`, `name`, `kind`, `version`), its optional `tags`, and `fetch`:

```
x <- datom_get_set(conn, "study001-adam")
dm <- x$members[[1]]$fetch(conn)
```

`fetch` resolves whatever the pointer points at: a table member yields data, a set member yields another `datom_set`. **A link pins the version it was read at** – it is a citation, not a subscription, so it never drifts to the latest. Pass a connection scoped to the member's own project; same-project members resolve through the connection you already have.

Two reads of the same set are **not** `identical()`, because closures compare by environment. Compare `m[c("id", "tags")]` instead, or use `identical(a, b, ignore.environment = TRUE)`.

Integrity, and what is not rechecked

The stored payload is verified against the recorded `document_sha` **before it is parsed**, and a version that records no `document_sha` is an error rather than a skipped check. `data_sha` is not recomputed: it is the address the payload was fetched from, so it would catch nothing the byte hash did not, and it would refuse a payload written by a newer `datom`. Nothing in the payload is re-canonicalized – what you are shown is what was cited.

See Also

`datom_write_set()` to write one, `datom_member()` to declare a member, `datom_read()` for tables.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datom_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  member <- datom_member(
    conn, "dm", datom_history(conn, "dm")$version[1],
    tags = list(type = "input")
  )
  datom_write_set(conn, list(member),
    tags = list(description = "Example product"))

  x <- datom_get_set(conn, "example_product")
  print(x)

  # Resolve one member to its data. The link pins the version it was read at.
  print(head(x$members[[1]]$fetch(conn)))

  unlink(tmp, recursive = TRUE)
}
```

datom_history	<i>Show Version History</i>
---------------	-----------------------------

Description

Returns the versions of a table or set, newest first (the 10 most recent by default): when each was saved, by whom, and with what message. Pass a value from the version column to [datom_read\(\)](#), or to [datom_get_set\(\)](#) for a set, to read that version back.

Usage

```
datom_history(conn, name, n = 10, short_hash = FALSE)
```

Arguments

conn	A <code>datom_conn</code> object from <code>datom_get_conn()</code> .
name	Table name.
n	Maximum number of versions to return. Default 10.
short_hash	If TRUE, truncates version and data SHA columns to 8 characters for readability. Default FALSE, so the version column can be passed straight to <code>datom_read()</code> .

Value

Data frame with columns: version, data_sha, timestamp, author, commit_message, commit_sha.

A version is content, not code

A `datom` version answers one question: *is this the same content and declared metadata?* Nothing code-derived enters it. So **a change that alters no content mints no new version**, and this is the behaviour most often reported as a bug.

Concretely: you refactor your build script, re-run it, and get byte-identical data. The write is a no-op, `datom_history()` shows the same version it showed before, and its `commit_sha` still points at the **earlier** commit – the one that first produced that content, which does not contain the code you are looking at. That is the recorded value doing its job. It names a commit that provably produces the version; it does not name every commit that could.

The commit is deliberately not part of the version. A set exists to be cited, and if a comment fix minted a new product version, "v47" would stop meaning anything.

Where `commit_sha` comes from

It is **derived, never authored** – no argument anywhere sets it. The copy in your clone does not carry it and cannot: `version_history.json` is committed *inside* the commit that would name it. Only the copy in storage has it, and it is there for readers who have no clone. With a clone, `git log -p {name}/set.json` answers the same question directly.

NA means the value is not recorded and could not be worked out from this repo's git history – a shallow clone or rewritten history, typically, or a version written by a `datom` too old to record it.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
}
```

```

)
datom_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datom_get_conn(file.path(tmp, "repo"), store)

datom_write(conn, data = datom_example_data("dm"), name = "dm")
print(datom_history(conn, "dm"))

unlink(tmp, recursive = TRUE)
}

```

<code>datom_init_repo</code>	<i>Create a New datom Project</i>
------------------------------	-----------------------------------

Description

Run once to start a new [project](#). It creates the project folder (including an `input_files/` folder for files you want to bring in), sets up git, pushes a first commit to GitHub – creating the GitHub repository if `create_repo = TRUE` – and records the new, empty project in storage. The store must carry a GitHub token; to join a project that already exists, use [datom_clone\(\)](#) instead.

Usage

```

datom_init_repo(
  path = ".",
  project_name,
  store,
  create_repo = FALSE,
  repo_name = project_name,
  max_file_size_gb = 1000,
  mode = NULL,
  set = NULL,
  git_ignore = c(".Rprofile", ".Renv", ".Rhistory", ".Rapp.history", ".Rproj.user/",
    ".DS_Store", "*.csv", "*.tsv", "*.rds", "*.txt", "*.parquet", "*.sas7bdat", ".RData",
    ".RDataTmp", "*.html", "*.png", "*.pdf", ".vscode/", "rsconnect/"),
  .force = FALSE
)

```

Arguments

<code>path</code>	Path to the project folder. Defaults to current directory.
<code>project_name</code>	Project name, used for S3 namespace and git repo.
<code>store</code>	A <code>datom_store</code> object (from <code>datom_store()</code>). Must have role "developer" (i.e., <code>github_pat</code> provided).
<code>create_repo</code>	If TRUE, create a GitHub repo via API. Mutually exclusive with providing <code>data_repo_url</code> on the store.
<code>repo_name</code>	GitHub repo name when <code>create_repo = TRUE</code> . Defaults to <code>project_name</code> . Useful when the project name (e.g., "STUDY_001") isn't a good GitHub repo name.

max_file_size_gb	Maximum file size limit in GB. Default 1000 (1TB).
mode	Project mode, or NULL (the default) for an ordinary data repo that onboards source files. The only other accepted value is "product", which declares a repo that builds its artifacts instead: it holds one set, writes derived tables, and refuses the file-import path. Absent is not a missing value here – it <i>is</i> "ordinary data repo", which is why nothing is written to project.yaml unless you ask for a product.
set	Name of the set a mode = "product" repo owns. Required with mode = "product" and refused without it: one repo holds one set, and a product repo that names none passes the mode check and then fails every set write, which is a repo that looks initialised and is not.
git_ignore	Character vector of patterns to add to .gitignore.
.force	If TRUE, skip the storage namespace safety check. Use only for intentional takeover of an existing namespace. Default FALSE. Two cases it does not cover, both refusals that stand: • a namespace that cannot be reached, because the manifest upload later in this function needs the same storage, so skipping the check buys nothing; • a mode = "product" repo, whose namespace check has no override at all – passing .force there is an error rather than a no-op, since a dropped override leaves you believing you took a namespace over when you did not.

Details

Initializes the **data repository only**. The project is left as a solo project: project.yaml is the location authority, no governance.json / dispatch.json / ref.json is written, and project.yaml omits the storage.governance and repos.governance blocks. A governance store component on store, if present, is ignored here. Governance is attached later via the governance layer (gov_at_attach()).

Value

Invisible TRUE on success.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
}
```

```

    )

    datum_init_repo(
      path = file.path(tmp, "repo"),
      project_name = "example_project",
      store = store
    )
    print(list.files(file.path(tmp, "repo"), all.files = TRUE, no.. = TRUE))

    unlink(tmp, recursive = TRUE)
  }

```

datum_lineage_union Union and Deduplicate source_lineage Lists

Description

Takes a list of zero or more `source_lineage` lists and returns their deduplicated union. Each entry is a list with `project`, `table`, and `version_sha`. Deduplication uses the composite key `paste(project, table, version_sha, sep = "\t")`, so each distinct entry appears exactly once and retained entries are returned unchanged.

Usage

```
datum_lineage_union(lineages)
```

Arguments

<code>lineages</code>	A list of <code>source_lineage</code> lists (each itself a list of entries with <code>project</code> , <code>table</code> , <code>version_sha</code>). NULL members are treated as empty.
-----------------------	--

Details

NULL members are tolerated (a parent may carry `source_lineage = NULL`) and treated as an empty contribution. Empty input, or a list containing only empty lineage lists, returns an empty list.

This helper is the building block of the composable lineage recompute recipe. To check that a derived table's recorded `source_lineage` matches its parents, read each parent through a connection scoped to that parent's project and union their lineages:

```

# conn_c is scoped to the derived table's project.
parents <- datum_get_parents(conn_c, "c")

# One connection per project, keyed by each parent's `source`. Never
# reach across project stores with a single connection.
conns <- list(project_a = conn_a, project_b = conn_b)

# Read each parent's lineage through its own project connection.
parent_sls <- lapply(parents, function(p) {

```

```

    datum_get_lineage(conns[[p$source]], p$table, version = p$version,
                      depth = "source")
  })

  recomputed <- datum_lineage_union(parent_sls)
  recorded    <- datum_get_lineage(conn_c, "c", depth = "source")
  identical(recomputed, recorded)

```

Value

A deduplicated list of source_lineage entries, or an empty list when there is nothing to union.

Examples

```

sl1 <- list(list(project = "p", table = "t", version_sha = "a"))
sl2 <- list(list(project = "p", table = "t", version_sha = "a"))
datum_lineage_union(list(sl1, sl2))

```

datum_list

List the Tables and Sets in a Project

Description

Returns one row per table and set in the project, with its current version and when it was last updated. Works with both developer and reader connections.

Usage

```
datum_list(conn, pattern = NULL, include_versions = FALSE, short_hash = TRUE)
```

Arguments

conn	A datum_conn object from datum_get_conn() .
pattern	Optional glob pattern for filtering table names.
include_versions	If TRUE, includes version count info.
short_hash	If TRUE (default), truncates version and data SHA columns to 8 characters for readability. Set to FALSE for full hashes.

Value

Data frame with artifact info (name, kind, current_version, last_updated, etc.).

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  print(datom_list(conn))

  unlink(tmp, recursive = TRUE)
}
```

datom_list_members	<i>List a Set's Members and Their Labels</i>
--------------------	--

Description

A data frame with one row per member **per label value** – long format, not wide. Tags are open-keyed and multi-valued, so a wide frame would need a list-column and a column set that changes from one set to the next; long is a plain frame with fixed columns whatever the set holds.

Usage

```
datom_list_members(x)
```

Arguments

x A `datom_set` from `datom_get_set()`.

Details

Filtering is therefore ordinary R – `subset()`, `dplyr::filter()` – and `datom` grows no query vocabulary of its own.

Value

A data frame of name, project, version, kind, key, value. Zero rows, with those columns, for a set with no members.

The columns

name, project, version and kind identify the member; key and value are one label. **An untagged member still gets a row**, with NA for both, so `unique(m$name)` is the complete member list rather than the tagged part of it.

value is a plain character column, never a list-column, because the tag grammar is text only.

See Also

[datum_structure_members\(\)](#) for a navigable view of the same labels, [datum_fetch_member\(\)](#) to resolve one member.

Examples

```
# These two facts about a set -- its members and their labels -- are all this
# verb reads, so a set built by hand shows the shape. In practice `x` comes
# from datum_get_set().
x <- structure(
  list(
    name = "study001-adam", project = "study001", version = NULL,
    data_sha = NULL, tags = list(description = "ADaM datasets"),
    members = list(
      list(
        id = list(project = "study001", name = "adsl", kind = "table",
          version = strrep("a", 64)),
        tags = list(type = "output", domain = c("safety", "efficacy"))
      ),
      list(
        id = list(project = "study001", name = "dm", kind = "table",
          version = strrep("b", 64))
      )
    )
  ),
  class = "datum_set"
)

# adsl appears three times, once per label; the untagged dm appears once.
datum_list_members(x)

# Filtering is plain R.
subset(datum_list_members(x), key == "domain" & value == "safety")
```

datum_member

Declare a Member of a Set

Description

Resolves one artifact version against a single project connection and returns a pure-data member record to pass to a set write. The record is a pointer: it names the project, artifact, kind, and version,

and carries no copy of the data. Reading the artifact's versioned metadata snapshot is what makes the pointer trustworthy – a member can only point at something that already exists, which is also why a set cannot contain itself at any depth.

Usage

```
datom_member(conn, name, version, tags = NULL)
```

Arguments

<code>conn</code>	A <code>datom_conn</code> scoped to the member's project store, from <code>datom_get_conn()</code> .
<code>name</code>	Artifact name (single validated string).
<code>version</code>	The artifact version (<code>metadata_sha</code>) to pin, e.g. from <code>datom_history()</code> .
<code>tags</code>	Optional named list of text labels for this member. Omitted from the record when absent or empty.

Details

Same-project and cross-project members are declared identically; the only difference is which connection is passed. `kind` comes from the snapshot (defaulting to "table" for a snapshot written before `datom` recorded the field), and `project` comes from the **repo** rather than from the connection: the artifact's own metadata, else the project manifest, else the connection's name with a warning saying it is unverified. That matters because a reader connection's project name is a label the caller supplied and nothing checks it against the repo, while this value is hashed into the set's identity and cited afterwards.

Unlike `datom_parent()`, a member carries **no** `data_sha`: the version already pins the content, and a second copy of that fact would be a second thing to keep consistent.

Value

A list with `id` (a list of exactly `project`, `name`, `kind`, `version`) and, when tags were supplied, `tags`. Pure data: it retains no connection and is serializable.

Tags

`tags` is an optional named list of text labels describing this member's role in the set – what folder structure would otherwise express. A value may be a single string or several, because the whole point of labels over folders is that an item can be in more than one category at once: `list(type = "output", domain = c("safety", "efficacy"))`.

Values are text only: no numbers, booleans, or nesting. Write a numeric label as a string ("500") and parse it downstream, exactly as you would a folder name.

Three outcomes, and the difference is whether anything is actually there:

What you pass	What happens
no tags	accepted; the record carries no tags
<code>list(domain = character(0))</code> or <code>list(domain = NULL)</code>	the key is dropped, as if never mentioned
<code>list(domain = "")</code>	refused – a label with no name is almost always an accident

See Also

[datom_parent\(\)](#) for the lineage equivalent.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")

  # Pin the version just written and label its role in the set.
  version <- datom_history(conn, "dm")$version[1]
  print(datom_member(conn, "dm", version, tags = list(type = "input")))

  unlink(tmp, recursive = TRUE)
}
```

datom_parent

Name an Input for a Table You Are About to Write

Description

Use before [datom_write\(\)](#) when the table you are writing was made from other datom tables. Each call names one input – one table at one exact version – and returns a note that [datom_write\(\)](#) saves with the new table, so its [lineage](#) records what it was made from. For several inputs, make one call each and pass them together as a list to parents. If the inputs are members of a set, pass the set as `x` and name several tables at once; each gets the version the set pins.

Usage

```
datom_parent(conn, table, version = NULL, x = NULL, tags = NULL)
```

Arguments

<code>conn</code>	A <code>datom_conn</code> scoped to the parent's project store, from <code>datom_get_conn()</code> .
<code>table</code>	Parent table name (single non-empty validated string). With <code>x</code> , a character vector of one or more member names.
<code>version</code>	Parent version (<code>metadata_sha</code> ; single non-empty string). Exactly one of <code>version</code> and <code>x</code> is required.
<code>x</code>	Optional <code>datom_set</code> from <code>datom_get_set()</code> (or one being built) whose members supply the versions. Exactly one of <code>version</code> and <code>x</code> is required.
<code>tags</code>	Optional named list of labels narrowing a member name the set holds more than once, e.g. <code>list(type = "input")</code> . Only with <code>x</code> .

Details

The parent's data fingerprint is read from the parent's own saved record; you cannot supply it, so a lineage entry cannot claim data the parent never had. The record is plain data with no connection inside, so it can be saved and reused.

Same-project and cross-project parents are declared identically – the only difference is which connection is passed. `source` is always derived from the connection's `project_name`.

Value

Without `x`, a list with exactly `source`, `table`, `version`, `data_sha`, and `source_lineage`. `source` is the project the parent's own metadata says it belongs to, falling back to the project manifest and then to the connection's name (see `.datom_declared_project()`) – **not** simply the name on `conn`, which on a reader connection is an unverified label and which source cannot afford, since it is part of the declaring table's version. `source_lineage` is `NULL` when the snapshot carries none. With `x`, an unnamed list of such records, one per table, in the order given.

Taking the version from a set

When the inputs of a derivation are members of a set, pass the set as `x` instead of a version: each table is looked up among the set's members and declared at the version the set pins. That keeps the parents a derived table records in step with the set it is derived through, without looking each version up by hand.

The member is chosen exactly as `datom_fetch_member()` chooses it. A name held by two members – a current table beside a locked baseline, say – stops and lists both; narrow with `tags`. A member that is itself a set stops too, since only a table can be a parent.

With `x`, `table` may name several tables and the result is **always a list** of parent records, even for one table, so it can be passed straight to the parents argument of `datom_write()`. Without `x` the result is one record, as it always was.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
```

```

remote <- file.path(tmp, "remote.git")
dir.create(remote, recursive = TRUE)
git2r::init(remote, bare = TRUE)

store <- datom_store(
  data = datom_store_local(file.path(tmp, "storage")),
  github_pat = "example-token", # role selector; a local remote needs none
  data_repo_url = remote,
  validate = FALSE
)
datom_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datom_get_conn(file.path(tmp, "repo"), store)

datom_write(conn, data = datom_example_data("dm"), name = "dm")

# Resolve a parent declaration to pass to the parents argument of
# datom_write.
print(datom_parent(conn, "dm", datom_history(conn, "dm")$version[1]))

unlink(tmp, recursive = TRUE)
}

```

datom_projects

List Projects Registered in the Governance Repo

Description

Returns a data frame with one row per project registered in the shared governance repo. Useful for managers and auditors who need to see the portfolio without having to clone every data repo.

Usage

```
datom_projects(x)
```

Arguments

x A `datom_conn` or a `datom_store` with a governance component.

Details

Accepts either a `datom_conn` (typically the developer's existing connection – reads the local gov clone) or a `datom_store` (lets a caller enumerate the portfolio before connecting to any specific project).

Read path:

- If a local gov clone is available (developer or any caller whose `gov_local_path` exists on disk), `projects/` is listed from disk and each `ref.json` is read locally. No network calls.
- Otherwise the gov storage client is used: `projects/` is listed and each `projects/{name}/ref.json` is fetched.

Corrupt registry entries (missing `ref.json`, unreadable JSON) emit a warning and are skipped – one bad project does not take down the listing.

Value

A data frame, sorted by name, with columns: `name` (character), `data_backend` (character), `data_root` (character), `data_prefix` (character; NA when absent), `registered_at` (character ISO8601 from clone mtime; NA on storage path).

Examples

```
# A governance store backed by a local directory. Projects are registered
# into it by the companion governance package (datomanager), so a freshly
# created governance store lists an empty portfolio.
tmp <- tempfile("datom-example-")
gov <- datom_store_local(file.path(tmp, "gov-storage"))

store <- datom_store(
  governance = gov,
  data       = datom_store_local(file.path(tmp, "storage")),
  gov_repo_url = "https://github.com/example/acme-gov"
)

datom_projects(store)

unlink(tmp, recursive = TRUE)
```

datom_pull

Pull Latest Changes from Remote

Description

Fetches and merges the latest git changes from the remote repository. This is the recommended entry point at the start of each work session to ensure the local state is current before syncing or writing tables.

Usage

```
datom_pull(conn)
```

Arguments

`conn` A `datom_conn` object from `datom_get_conn()`.

Details

Git is the source of truth for all metadata (manifest, dispatch, table metadata). The manifest and other metadata files live in git and are pulled along with any other committed changes.

Requires developer role (readers have no git access).

Value

Invisibly, a list with:

commits_pulled Integer count of new commits merged.

branch Current branch name.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Nothing new on the remote yet, so this is a no-op.
  datom_pull(conn)

  unlink(tmp, recursive = TRUE)
}
```

datom_read

Read a datom Table

Description

Returns a table as a data frame: the current version by default, or a past version when you pass version (copy it from [datom_history\(\)](#)). Works with both developer and reader connections. To read a set, use [datom_get_set\(\)](#).

Usage

```
datom_read(conn, name, version = NULL, context = NULL, ...)
```

Arguments

conn	A datom_conn object from datom_get_conn() .
name	Table name.

version	Optional metadata_sha (datom version). If NULL, uses current.
context	Reserved; currently ignored.
...	Reserved; currently ignored.

Value

A data frame.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")

  # Current version
  dm <- datom_read(conn, "dm")
  print(head(dm))

  # A specific version, by its identifier -- byte-for-byte the same table
  v <- datom_history(conn, "dm")$version[1]
  print(identical(datom_read(conn, "dm", version = v), dm))

  unlink(tmp, recursive = TRUE)
}
```

datom_remove_members Drop Members from a Set

Description

Removes the members you select and returns the set without them. **Nothing is written:** the object comes back edited, and the set is stored only when you pass the result to [datom_write_set\(\)](#).

Usage

```
datom_remove_members(x, member = NULL, tags = NULL, version = NULL)
```

Arguments

x	A <code>datom_set</code> , from <code>datom_get_set()</code> or <code>datom_assemble_set()</code> .
member	The member to drop, as its name, a member record, or a link. Optional only when tags or version selects on its own.
tags	Optional named list of labels selecting members, e.g. <code>list(status = "draft")</code> . A member matches when it carries every label listed.
version	Optional version, or a prefix of one, selecting the members pinned at it.

Details

It exists because the hand-rolled version is silently wrong. Filtering a member list by name drops **every** version of that name, so a set holding a live table beside a deliberately frozen baseline loses both; removing by position removes a different member the day somebody adds one.

Value

x without the selected members, its `version` and `data_sha` emptied because they described a payload it no longer holds, and what was dropped appended to its `datom_edits` attribute – which `datom_write_set()` turns into the commit message.

Why there is no connection argument

Not an oversight. Removing a member only has to **find** a pointer the set already holds, while adding or repointing one has to **resolve** it – read the artifact’s metadata, confirm its kind, record the project that wrote it. So this verb does no IO at all and needs no credentials, which is also why it is the one edit verb that works on a set read through a storage-only connection with no clone.

Selecting what to drop

A selection is **required**. `datom_remove_members(x)` would mean removing every member, which the writer refuses anyway, so it aborts instead of building a payload the write then rejects. The safe default for a destructive verb is nothing – the opposite of `datom_update_members()`, where the safe default is everything because a refresh is idempotent.

Select by name, by a member record, or by a link, and narrow with tags or version; tags or version on their own select every member they match, so `tags = list(status = "draft")` drops the labelled ones.

Three things it refuses rather than doing quietly:

What you asked for	Why it stops
a name matching more than one member	that is the silently-wrong spelling this verb replaces – it would drop a frozen baseline
a selection matching nothing	it is a typo, and <code>Filter()</code> reports success for it
every member	a set with no members cannot be written, and the refusal belongs on the line that erases

An ambiguous name refuses here while `datom_update_members()` skips it, and the asymmetry is the consequence rather than a taste: skipping a repoint leaves a valid pinned version behind, while skipping a removal silently does nothing at all.

See Also

`datom_update_members()` to repoint members instead, `datom_list_members()` to see what a set holds, `datom_write_set()` to store the result.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datom_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  datom_write(conn, data = datom_example_data("lb"), name = "lb")
  datom_write_set(conn, list(
    datom_member(conn, "dm", datom_history(conn, "dm")$version[1],
      tags = list(type = "input")),
    datom_member(conn, "lb", datom_history(conn, "lb")$version[1],
      tags = list(status = "draft"))
  ))

  x <- datom_get_set(conn, "example_product")

  # By label, which is the selection that does not depend on position.
  x <- datom_remove_members(x, tags = list(status = "draft"))
  print(datom_list_members(x))

  datom_write_set(conn, x)

  unlink(tmp, recursive = TRUE)
}
```

 datum_repo_attach_governance

Write the Data-Side Governance Attachment Record

Description

Writes `governance.json` – the data-side pointer recording which governance repository a project is attached to. This is the data-repo / data-storage half of attaching governance; the gov-repo registration (writing `ref.json` and `dispatch.json`, committing to the gov repo) is performed separately by the governance layer (`datamanager::gov_attach()`).

Usage

```
datum_repo_attach_governance(conn, gov_repo_url, gov_store, message = NULL)
```

Arguments

<code>conn</code>	A <code>datum_conn</code> object with <code>role = "developer"</code> and a local data clone (<code>conn\$path</code>).
<code>gov_repo_url</code>	HTTPS clone URL of the governance git repository to record.
<code>gov_store</code>	A <code>datum_store_s3</code> or <code>datum_store_local</code> component for the governance storage. Only its location fields are persisted; credentials are discarded.
<code>message</code>	Optional commit message. Defaults to <code>"Attach governance: {project_name}"</code> .

Details

`governance.json` is the canonical data->gov pointer in the bidirectional governance link: the gov repo's `ref.json` points gov->data, and this file points data->gov, so either repo can find the other. It is written to two locations, mirroring the manifest pattern (git canonical, storage derived):

- `.datum/governance.json` in the local data clone (git canonical), committed and pushed to the data repo.
- `{prefix}/datum/.metadata/governance.json` in data storage (derived mirror; a failed mirror write warns but does not abort – the git copy is canonical and readers with gov access resolve location from the gov repo).

Routing this write through datum upholds the two-repos invariant: the governance layer never mutates the data repo directly.

Value

Invisibly, the SHA of the resulting data-repo commit.

See Also

[datum_repo_delete\(\)](#), [datum_repo_set_data_store\(\)](#)

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Data-side half of governance attachment. The gov-repo registration
  # is performed separately by the companion datomanager package.
  datom_repo_attach_governance(
    conn,
    gov_repo_url = "https://github.com/example/acme-gov",
    gov_store = datom_store_local(file.path(tmp, "gov-storage"))
  )
  gov_json <- jsonlite::read_json(
    file.path(tmp, "repo", ".datom", "governance.json")
  )
  print(gov_json$gov_repo_url)

  unlink(tmp, recursive = TRUE)
}
```

datom_repo_commit	<i>Commit Content in the Data Repo</i>
-------------------	--

Description

Commits changes in the data repo clone and, by default, pushes them. This is datom's **sanctioned git-mutation surface** for downstream packages: a build or deployment package commits its own content – code, `renv.lock`, framework state – through this verb rather than importing `git2r` and writing to the data repo behind datom's back.

Usage

```
datom_repo_commit(conn, message, paths = NULL, push = TRUE)
```

Arguments

conn	A datum_conn object with role = "developer" and a local repo path (conn\$path).
message	Commit message. Required, and used only when a commit is actually created.
paths	NULL (default) to stage everything git add . would stage, or a character vector of repo-relative paths to stage exactly those. Explicit paths must exist: to record a deletion, use paths = NULL.
push	Push after committing (default TRUE), through the same path datum's own writes use – so it inherits pull-before-push and upstream tracking. push = FALSE commits only; datum_repo_push() is the other half of that split.

Details

paths = NULL **means what** git add . **means**. It stages tracked modifications, deletions and untracked files, minus anything .gitignore excludes. That is the correct semantic for a human-invoked moment, and it is deliberately the opposite of what datum's own writes do: a commit created inside datum_write() stages an explicit file list, because it fires at a moment datum chose and must never sweep up work in progress.

One consequence of add-all worth knowing rather than discovering: if an earlier datum write failed after writing local metadata but before committing, those datum files are dirty and this verb will stage them. That is left intentional – silently excluding datum's own paths would make the argument lie about its contract, and the state is exactly what datum_validate() reports and datum_validate(fix = TRUE) repairs. It also moves git *ahead* of storage, which is the safe direction.

Commit is idempotent, push is convergent, and neither implies the other. A clean tree produces no commit and is not an error, so "commit everything" can be called twice. With push = TRUE the push still runs when the branch is ahead of the remote, even though no commit was created – otherwise one failed push would leave the remote behind forever, since every later call finds a clean tree and returns early.

Value

Invisibly, the commit SHA; invisible(NULL) when no commit was created (whether or not a push happened).

See Also

datum_repo_push(), datum_validate()

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datum-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)
```

```

store <- datom_store(
  data = datom_store_local(file.path(tmp, "storage")),
  github_pat = "example-token", # role selector; a local remote needs none
  data_repo_url = remote,
  validate = FALSE
)
datom_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datom_get_conn(file.path(tmp, "repo"), store)

# Content datom does not own, committed through datom.
dir.create(file.path(tmp, "repo", "R"))
writeLines("build <- function() NULL", file.path(tmp, "repo", "R", "build.R"))
datom_repo_commit(conn, "Add build script")

# Idempotent: a clean tree is a no-op, not an error.
datom_repo_commit(conn, "Nothing to do")

unlink(tmp, recursive = TRUE)
}

```

datom_repo_delete

Delete the Data GitHub Repository and Local Clone

Description

Deletes the data-side GitHub repository via the GitHub REST API and removes the local clone directory. This is the data-side teardown step for a datom project.

Usage

```
datom_repo_delete(conn, confirm, force_gov_attached = FALSE)
```

Arguments

<code>conn</code>	A <code>datom_conn</code> object (developer role required).
<code>confirm</code>	Character string. Must equal <code>conn\$project_name</code> exactly. No interactive prompts – this must be supplied explicitly.
<code>force_gov_attached</code>	Logical. FALSE (default) refuses to run when governance is attached (<code>!is.null(conn\$gov_root)</code>). Pass TRUE only when called programmatically from <code>datomanager::gov_decommission()</code> .

Details

Solo projects (no governance attached): call this together with [datom_storage_delete_prefix\(\)](#) for a complete teardown.

Governed projects: use `datomanager::gov_decommission()` instead. That function calls `datom_repo_delete()` internally (with `force_gov_attached = TRUE`). Calling `datom_repo_delete()` directly on a governed project without that flag is refused to prevent accidentally orphaning the governance registration.

Steps:

1. Delete the data GitHub repo via the GitHub REST API (requires conn\$github_pat with delete_repo scope; skipped with a warning when conn\$github_pat is NULL or when the remote is not GitHub). Aborts if conn\$data_repo_url is not set.
2. Remove the local clone directory (conn\$path).

Each step is warn-and-continue on failure so the other still runs.

Value

Invisible TRUE on success.

See Also

[datom_storage_delete_prefix\(\)](#), [datom_repo_set_data_store\(\)](#)

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage. Because the remote is not GitHub,
# the API deletion step is skipped and only the local clone is removed.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Solo project teardown (no governance): storage, then repo.
  datom_storage_delete_prefix(conn)
  datom_repo_delete(conn, confirm = conn$project_name)

  unlink(tmp, recursive = TRUE)
}
```

datom_repo_push

Push the Data Repo to Its Remote

Description

Pushes the current branch of the data repo clone, through the same path datom's own writes use – so it inherits pull-before-push, upstream tracking, and the on-a-branch guard.

Usage

```
datom_repo_push(conn)
```

Arguments

conn A `datom_conn` object with role = "developer" and a local repo path (`conn$path`).

Details

Convergent, not imperative. Nothing to push is an informational no-op rather than an error, so calling it twice is safe and "make sure the remote has everything" is a legal standalone operation.

This is the other half of `datom_repo_commit()` (`push = FALSE`). Without it, "push what I already committed" would only be expressible as *another commit attempt* – and since `paths = NULL` is add-all, a caller who merely wanted to push would risk committing whatever work in progress the tree happened to hold.

Value

Invisibly TRUE.

See Also

[datom_repo_commit\(\)](#), [datom_pull\(\)](#)

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Commit now, push later.
  writeLines("notes", file.path(tmp, "repo", "NOTES.md"))
  datom_repo_commit(conn, "Add notes", push = FALSE)
  datom_repo_push(conn)

  # Convergent: a second call has nothing to do and says so.
  datom_repo_push(conn)
```

```
    unlink(tmp, recursive = TRUE)
}
```

datum_repo_set_data_store

Rewrite the Data Store Pointer in project.yaml

Description

Updates `storage.data` in `.datum/project.yaml` to point at `new_store`, then commits and pushes the data repo. This is the data-side bookkeeping step of a store relocation.

Usage

```
datum_repo_set_data_store(conn, new_store, message = NULL)
```

Arguments

<code>conn</code>	A <code>datum_conn</code> object with <code>role = "developer"</code> and a local repo path (<code>conn\$path</code>).
<code>new_store</code>	A <code>datum_store_s3</code> or <code>datum_store_local</code> component (i.e. the data-side component of a <code>datum_store()</code> object, not the full composite).
<code>message</code>	Optional commit message. Defaults to <code>"Update data store: {project_name}"</code> .

Details

Read-modify-write contract: the function reads the full existing `project.yaml`, modifies **only** `storage.data`, and writes back. It never reconstructs the file from `conn` fields. This preserves `storage.governance` on governed projects (it is permanent once written) and any other fields not owned by this function.

For governed projects the authoritative address is `ref.json` in the gov repo – this function updates only the local data clone so that `datum_get_conn()` stays consistent after migration. It is called by `datomanager::gov_migrate_data()` after the ref switch, never before.

Value

Invisibly, the SHA of the resulting commit.

See Also

[datum_storage_copy\(\)](#), [datum_storage_verify\(\)](#), [datum_repo_delete\(\)](#)

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Reprint project.yaml at a relocated data store.
  new_store <- datom_store_local(file.path(tmp, "storage-relocated"))
  datom_repo_set_data_store(conn, new_store)

  unlink(tmp, recursive = TRUE)
}
```

datom_status

Show Repository Status

Description

Displays connection info, table count, and (for developers) uncommitted git changes and input file sync state.

Usage

```
datom_status(conn)
```

Arguments

conn A `datom_conn` object from `datom_get_conn()`.

Value

Invisibly, a list with connection, tables, and optionally git and input_files status details.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_status(conn)

  unlink(tmp, recursive = TRUE)
}
```

datom_storage_copy	<i>Copy All Objects Between Two datom Storage Namespaces</i>
--------------------	--

Description

Enumerates all objects under from_conn's datom namespace and streams each one to to_conn's datom namespace. All four backend combinations are supported:

Usage

```
datom_storage_copy(from_conn, to_conn)
```

Arguments

from_conn	A datom_conn object (source).
to_conn	A datom_conn object (destination).

Details

- **local -> local:** direct file copy via `fs::file_copy()`.
- **local -> S3:** reads raw bytes and uploads via `put_object`.
- **S3 -> local:** downloads via `get_object` and writes to disk.

- **S3 -> S3:** streams bytes through memory (get then put). Server-side copy_object (same-region optimisation) is reserved for a future release.

This is a policy-free primitive. It does not modify the source namespace, update `project.yaml`, or switch `ref.json`. For a complete managed migration (governed projects) use `datomanager::gov_migrate_data()`. For solo-project relocation combine this function with `datom_repo_set_data_store()`.

Value

A data frame with columns `key` (character, relative key after {prefix}/datom/) and `bytes` (numeric, byte count per object). Returns a zero-row data frame if the source namespace is empty.

See Also

`datom_storage_verify()`, `datom_storage_list()`, `datom_storage_delete_prefix()`, `datom_repo_set_data_store()`

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and two
# local directories for the source and destination object stores.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  from_conn <- datom_get_conn(file.path(tmp, "repo"), store)
  datom_write(from_conn, data = datom_example_data("dm"), name = "dm")

  # The destination is addressed with a reader connection: no local repo,
  # just a store plus the project name.
  to_store <- datom_store(data = datom_store_local(file.path(tmp, "storage2")))
  to_conn <- datom_get_conn(store = to_store, project_name = "example_project")

  copied <- datom_storage_copy(from_conn, to_conn)
  print(nrow(copied))      # number of objects copied
  print(sum(copied$bytes)) # total bytes

  unlink(tmp, recursive = TRUE)
}
```

datum_storage_delete_prefix

Delete All Objects Under a datum Storage Prefix

Description

Removes every file under {prefix}/datum/{prefix_key} from storage. Pass prefix_key = NULL (the default) to delete the entire datum namespace for this connection. A missing or empty prefix is a no-op.

Usage

```
datum_storage_delete_prefix(conn, prefix_key = NULL)
```

Arguments

conn	A datum_conn object.
prefix_key	Relative prefix to delete under (after {prefix}/datum/). NULL (default) deletes the entire datum namespace root for this connection.

Details

Irreversible. Intended for package developers building tools on top of datum (e.g. datomanager for rollback or source deletion after migration). End users performing a full project teardown should use [datum_repo_delete\(\)](#) instead.

Value

Invisibly, a backend-specific value. For S3: the count of deleted objects (0L if nothing found). For the local backend: 1L if the prefix directory existed and was removed, 0L otherwise.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datum-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datum_store(
    data = datum_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datum_init_repo(file.path(tmp, "repo"), "example_project", store)
```

```

conn <- datom_get_conn(file.path(tmp, "repo"), store)
datom_write(conn, data = datom_example_data("dm"), name = "dm")

# Delete a single table's objects
datom_storage_delete_prefix(conn, prefix_key = "dm")

# Delete the entire datom namespace (use with care)
datom_storage_delete_prefix(conn)
print(datom_storage_list(conn))

unlink(tmp, recursive = TRUE)
}

```

<code>datom_storage_list</code>	<i>List All Objects in a datom Storage Namespace</i>
---------------------------------	--

Description

Returns the full storage keys of every object under the datom namespace for this connection (`{prefix}/datom/...`). Intended for package developers building tools on top of datom (e.g. `datomanager`); end users typically do not need to inspect raw storage keys directly.

Usage

```
datom_storage_list(conn)
```

Arguments

<code>conn</code>	A <code>datom_conn</code> object.
-------------------	-----------------------------------

Details

Keys are returned in their full storage-key form – for S3 that is `"{prefix}/datom/..."` relative to the bucket root; for local backends it is a path relative to `conn$root`. This mirrors the contract of the internal `.datom_storage_list_objects()` dispatch layer.

Value

A character vector of full storage keys. May be empty if the namespace contains no objects.

Examples

```

# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)
}

```

```

store <- datum_store(
  data = datum_store_local(file.path(tmp, "storage")),
  github_pat = "example-token", # role selector; a local remote needs none
  data_repo_url = remote,
  validate = FALSE
)
datum_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datum_get_conn(file.path(tmp, "repo"), store)
datum_write(conn, data = datum_example_data("dm"), name = "dm")

print(datum_storage_list(conn))

unlink(tmp, recursive = TRUE)
}

```

datum_storage_read_json

Read a JSON Document from a datum Storage Namespace

Description

Reads and parses a JSON object from storage, dispatching on the connection's backend. Intended for package developers building tools on top of datum (e.g. `datomanager`) that need to inspect datum's own documents – manifests, metadata, version history – without reaching into internals via `...`.

Usage

```
datum_storage_read_json(conn, key)
```

Arguments

<code>conn</code>	A <code>datum_conn</code> object.
<code>key</code>	Relative storage key (after <code>{prefix}/datum/</code>).

Details

End users should prefer the purpose-built readers: `datum_read()` for table data, `datum_list()` and `datum_summary()` for the manifest, and `datum_history()` for version history. This export is a byte-level primitive and does not interpret what it reads.

`key` is a **relative** key – the portion after `{prefix}/datum/`, e.g. `"dm/.metadata/metadata.json"`. The backend prepends the namespace itself, so passing a full key (one containing a `datum/` segment) is refused rather than silently resolving under `{prefix}/datum/{prefix}/datum/` and finding nothing. Keys containing a `.` segment or a leading `/` are refused too: reads are confined to this project's namespace.

There is no corresponding write export. Documents are written by purpose-built verbs (`datum_write()`, `datum_repo_attach_governance()`), not by a generic byte channel, so that every mutation of a datum namespace routes through a function that knows what it is writing.

Value

The parsed JSON document as an R list. Nested structures are kept as lists (`simplifyVector = FALSE`), so JSON arrays of objects do not collapse into data frames.

See Also

`datom_storage_list()` to discover keys; `datom_read()`, `datom_list()`, `datom_history()` for the interpreted equivalents.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)
  datom_write(conn, data = datom_example_data("dm"), name = "dm")

  # Relative key: the part after `{prefix}/datom/`
  meta <- datom_storage_read_json(conn, "dm/.metadata/metadata.json")
  print(meta$hash_algo)

  unlink(tmp, recursive = TRUE)
}
```

`datom_storage_verify` *Verify a Copy Between Two datom Storage Namespaces*

Description

Checks that objects in `to_conn`'s `datom` namespace match their counterparts in `from_conn`. Two verification modes are available:

Usage

```
datom_storage_verify(
  from_conn,
  to_conn,
```

```

    keys = NULL,
    mode = c("structural", "content")
  )

```

Arguments

from_conn	A <code>datom_conn</code> object (source / reference).
to_conn	A <code>datom_conn</code> object (destination to verify).
keys	Character vector of relative keys (after {prefix}/datom/) to verify. NULL (default) verifies every key returned by <code>datom_storage_list(from_conn)</code> . Pass a subset to verify a sample.
mode	"structural" (default) or "content". See above.

Details

- **"structural" (default):** Confirms each destination object exists and its byte size matches the source. Fast – one HEAD/stat per object, no byte transfer. Catches truncated or missing objects, which is the dominant copy failure mode.
- **"content":** Re-reads destination bytes, recomputes the SHA-256 hash, and compares against the source hash. Expensive (full re-download for remote backends) but gives true bit-level integrity. Use for regulated or paranoid runs.

Value

A data frame with columns:

- `key` (character): relative storage key.
- `ok` (logical): TRUE if the object passed verification.
- `issue` (character): description of the mismatch, or NA if ok. Returns a zero-row data frame if keys is empty.

See Also

[datom_storage_copy\(\)](#), [datom_storage_list\(\)](#)

Examples

```

# Offline, self-contained: a bare git repo stands in for GitHub and two
# local directories for the source and destination object stores.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,

```

```

    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  from_conn <- datom_get_conn(file.path(tmp, "repo"), store)
  datom_write(from_conn, data = datom_example_data("dm"), name = "dm")

  to_store <- datom_store(data = datom_store_local(file.path(tmp, "storage2")))
  to_conn <- datom_get_conn(store = to_store, project_name = "example_project")
  copied <- datom_storage_copy(from_conn, to_conn)

  # Verify all copied objects structurally (default, fast)
  results <- datom_storage_verify(from_conn, to_conn)
  print(all(results$ok))

  # Verify a subset with full content hash
  print(datom_storage_verify(from_conn, to_conn,
                             keys = copied$key[1],
                             mode = "content"))

  unlink(tmp, recursive = TRUE)
}

```

datom_store

Create a datom Store

Description

A store tells datom where a project's data is kept – a local folder ([datom_store_local\(\)](#)) or an S3 bucket ([datom_store_s3\(\)](#)) – and, if you will be writing, your GitHub token. With a token you are a [developer](#) and can write; without one you are a reader and can only read. Pass the store to [datom_init_repo\(\)](#) to start a project, or to [datom_get_conn\(\)](#) to connect to one.

Usage

```

datom_store(
  governance = NULL,
  data,
  github_pat = NULL,
  data_repo_url = NULL,
  gov_repo_url = NULL,
  gov_local_path = NULL,
  github_org = NULL,
  github_api_url = NULL,
  validate = TRUE
)

```

Arguments

<code>governance</code>	A store component (e.g., <code>datom_store_s3()</code>) for governance files (dispatch, ref, migration history), or <code>NULL</code> for a no-governance store. A no-governance store represents a project that has not yet been promoted to governance (via the <code>datomanager</code> package); <code>gov_repo_url</code> and <code>gov_local_path</code> must also be <code>NULL</code> in that case.
<code>data</code>	A store component (e.g., <code>datom_store_s3()</code>) for data files (manifest, tables, metadata).
<code>github_pat</code>	GitHub personal access token. If provided, role is "developer". If <code>NULL</code> , role is "reader".
<code>data_repo_url</code>	GitHub remote URL for the data repository. Required when <code>github_pat</code> is provided and <code>create_repo = FALSE</code> in <code>datom_init_repo()</code> .
<code>gov_repo_url</code>	GitHub remote URL for the shared governance repository. The governance repo is created once per org (via the <code>datomanager</code> package) and referenced here by every project that uses it.
<code>gov_local_path</code>	Local directory path for the governance clone. If <code>NULL</code> (default), the clone is placed as a sibling of the data repo, named after the basename of <code>gov_repo_url</code> (e.g., "acme-gov").
<code>github_org</code>	GitHub organization for repo creation. <code>NULL</code> for personal repos.
<code>github_api_url</code>	GitHub API base URL. <code>NULL</code> (default) uses "https://api.github.com", which is correct for github.com and GitHub Enterprise Cloud (GHEC). For GitHub Enterprise Server (GHES) pass the server's API root, e.g. "https://github.mycompany.com/api/v3". A trailing / is stripped for consistency.
<code>validate</code>	If <code>TRUE</code> (default), validate GitHub PAT via API. Set to <code>FALSE</code> for tests or offline use.

Value

A `datom_store` object.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store(
  data = datom_store_local(path = tmp),
  data_repo_url = "https://github.com/example/my-project",
  validate = FALSE
)
store
is_datom_store(store)
unlink(tmp, recursive = TRUE)
```

datom_store_local	Create a Local Filesystem Store Component
-------------------	---

Description

Constructs a validated local filesystem storage component for use as either the governance or data component of a `datom_store`. Validates that the path exists (or is creatable) and is writable.

Usage

```
datom_store_local(path, prefix = NULL, validate = TRUE)
```

Arguments

path	Directory path for the store root.
prefix	Key prefix within the root (e.g., "project/"). NULL for no prefix.
validate	If TRUE (default), validate that path exists and is writable. Set to FALSE for tests or deferred creation.

Value

A `datom_store_local` object.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store_local(path = tmp, validate = TRUE)
store
is_datom_store_local(store)
unlink(tmp, recursive = TRUE)
```

datom_store_s3	Create an S3 Store Component
----------------	------------------------------

Description

Constructs a validated S3 storage component for use as either the governance or data component of a `datom_store`. Validates credentials and bucket access at construction time (unless `validate = FALSE`).

Usage

```

datom_store_s3(
  bucket,
  prefix = NULL,
  region = "us-east-1",
  access_key,
  secret_key,
  session_token = NULL,
  validate = TRUE
)

```

Arguments

bucket	S3 bucket name.
prefix	S3 key prefix (e.g., "project/"). NULL for no prefix.
region	AWS region (default "us-east-1").
access_key	AWS access key ID.
secret_key	AWS secret access key.
session_token	Optional AWS session token (for temporary credentials).
validate	If TRUE (default), validate credentials and bucket access at construction time. Set to FALSE for tests or offline use.

Value

A `datom_store_s3` object.

Examples

```

s3 <- datom_store_s3(
  bucket = "my-datom-bucket",
  prefix = "project/",
  region = "us-east-1",
  access_key = "AKIAIOSFODNN7EXAMPLE",
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
  validate = FALSE
)
s3
is_datom_store_s3(s3)

```

datom_store_s3_creds	<i>Create a Credentials-Only S3 Store Component</i>
----------------------	---

Description

Constructs an S3 store component that carries only AWS credentials – no bucket, prefix, or region. The data location is resolved at connection time from `ref.json` stored in the governance repo. This is the recommended construction style for readers when a governance store is in place.

Usage

```
datom_store_s3_creds(access_key, secret_key, session_token = NULL)
```

Arguments

access_key AWS access key ID.
secret_key AWS secret access key.
session_token Optional AWS session token (for temporary credentials).

Details

A `datom_store_s3_creds` component **must** be paired with a governance component inside `datom_store()`. Attempting to create a composite store without governance will abort with a clear message.

Value

A `datom_store_s3_creds` object.

Examples

```
creds <- datom_store_s3_creds(
  access_key = "AKIAIOSFODNN7EXAMPLE",
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY"
)
creds
is_datom_store_s3_creds(creds)
```

datom_structure_members

Group a Set's Members into a Navigable View

Description

Groups members by the values of the label key(s) named in `by` and returns a nested list whose leaves are the members' **links**, so `dp$output$ads1(conn)` works and tab-completes.

Usage

```
datom_structure_members(x, by, missing = "untagged")
```

Arguments

x A `datom_set` from `datom_get_set()`.
by Character vector of one or more label keys to group by, outermost first.
missing Branch name for members carrying no value for an axis key.

Details

A pure function of `x` and the axis you ask for: nothing is stored, and `datom` takes no position on which hierarchy is the right one. Ask for `by = c("domain", "type")` and you get a different tree from the same set, which is that design working rather than being worked around.

Value

A nested list `length(by) + 1` levels deep: one level per axis, then the member's own name holding its `datom_link`. An empty list for a set with no members.

One member, several branches

A member tagged `domain = c("safety", "efficacy")` appears under **both** `safety` and `efficacy`, so the total number of leaves can exceed the number of members. That is the point of labels over folders rather than a quirk of this verb: a folder holds an item in exactly one place, and a label does not.

What is refused, and why nothing is dropped

Two requests abort, because the alternative in both cases is a member the consumer cannot find and cannot see is absent:

- **Two members asking for one leaf name.** Two members may legitimately share a name at different versions, and if both carry the same label they ask for the same leaf. The abort names both and points at adding an axis – `by = c("type", "release")`. The set itself is entirely legal; only this projection of it is refused.
- **A missing bucket name that is also a real label value.** The bucket is a leaf name, so a set where some member genuinely carries `type = "untagged"` would merge the real branch into the bucket. Refused whatever the members happen to look like, so that whether the projection works does not depend on whether a member is currently missing the key.

A member that simply lacks the axis key is **not** refused and **not** dropped: it goes under `missing`, named, because a named bucket is visible and an omission is not.

See Also

`datom_list_members()` for the flat view, `datom_fetch_member()` to resolve one member by name.

Examples

```
# Built by hand to show the shape; in practice `x` comes from
# datom_get_set(). adsl carries two domains, so it appears under both.
x <- structure(
  list(
    name = "study001-adam", project = "study001", version = NULL,
    data_sha = NULL, tags = NULL,
    members = list(
      list(
        id = list(project = "study001", name = "adsl", kind = "table",
```

```

        version = strrep("a", 64)),
      tags = list(domain = c("safety", "efficacy"))
    ),
    list(
      id = list(project = "study001", name = "dm", kind = "table",
        version = strrep("b", 64))
    )
  ),
  class = "datom_set"
)

dp <- datom_structure_members(x, by = "domain")
print(names(dp))
print(names(dp$safety))

# A leaf is a link: call it with a connection to resolve it.
print(dp$safety$adsl)

```

datom_summary	<i>Summarize a datom Project</i>
---------------	----------------------------------

Description

Returns a compact, role-aware overview of a datom project: its name, backend, table/version totals, last write time, and (for developers) the git remote URL. Reads `.metadata/manifest.json` from the data store.

Usage

```
datom_summary(conn)
```

Arguments

`conn` A `datom_conn` object from [datom_get_conn\(\)](#).

Value

A `datom_summary` S3 object (a list with class `"datom_summary"`) containing: `project_name`, `role`, `backend`, `root`, `prefix`, `table_count`, `set_count`, `total_versions`, `last_updated`, `remote_url`. `table_count` counts tables only and `set_count` counts sets; `total_versions` stays tables-only, so no counter changed meaning. `remote_url` is NULL for readers (no local data clone).

Examples

```

# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")

```

```

remote <- file.path(tmp, "remote.git")
dir.create(remote, recursive = TRUE)
git2r::init(remote, bare = TRUE)

store <- datom_store(
  data = datom_store_local(file.path(tmp, "storage")),
  github_pat = "example-token", # role selector; a local remote needs none
  data_repo_url = remote,
  validate = FALSE
)
datom_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datom_get_conn(file.path(tmp, "repo"), store)

datom_write(conn, data = datom_example_data("dm"), name = "dm")
print(datom_summary(conn))

unlink(tmp, recursive = TRUE)
}

```

datom_sync

Bring New and Changed Files Into a Project

Description

Takes the preview from [datom_sync_manifest\(\)](#) and saves each new or changed file as a version of a table named after the file; unchanged files are skipped. This is the usual way to bring files into datom. To save a data frame you built in R, use [datom_write\(\)](#).

Usage

```

datom_sync(
  conn,
  manifest,
  continue_on_error = TRUE,
  sources = NULL,
  tags = list(type = "input"),
  x = NULL
)

```

Arguments

conn	A <code>datom_conn</code> object from datom_get_conn() .
manifest	Data frame from datom_sync_manifest() . On an ordinary repo, with columns <code>name</code> , <code>file</code> , <code>format</code> , <code>original_file_sha</code> , <code>status</code> . On a product repo, the set preview: <code>project</code> , <code>name</code> , <code>kind</code> , <code>version_from</code> , <code>version_to</code> , <code>status</code> . Any subset of its rows will do.

<code>continue_on_error</code>	If TRUE (default), continues processing remaining tables when one fails. If FALSE, stops on first error. Not accepted on a product repo, where one failure stops the call and nothing has been written.
<code>sources</code>	On a product repo only, and required there: one <code>datom_conn</code> , or a list of them, for the projects named by the rows being applied – the same connections the preview was built with. Refused on an ordinary repo.
<code>tags</code>	On a product repo only: the labels given to members added by new rows. Default <code>list(type = "input")</code> . A repointed member keeps its own labels. Refused on an ordinary repo.
<code>x</code>	On a product repo only: the <code>datom_set</code> to apply the preview to, from <code>datom_get_set()</code> or <code>datom_assemble_set()</code> . Omitted, the repo's stored set is read, or an empty one used when it has never been written. Refused on an ordinary repo.

Details

Reading files needs the `rio` package (`install.packages("rio")`).

Rows flagged "unsupported_format" by `datom_sync_manifest()` are reported as `result = "error"` with the recourse in the error column; the rest of the batch still processes.

On a product repo (mode: product) it applies a preview of the repo's set instead – see "On a product repo" below.

Value

On an ordinary repo, the manifest data frame augmented with `result` and `error` columns. `result` is "success", "skipped", or "error".

On a product repo, the updated `datom_set`, with what changed appended to its `datom_edits` attribute. Its `version` and `data_sha` are emptied when any row was applied.

On a product repo

A product repo owns one set, and this call applies a preview from `datom_sync_manifest()` to it: each new row adds a member at `version_to`, labelled with `tags`, and each changed row repoints the member it names from `version_from` to `version_to`, keeping that member's labels exactly. Rows of any other status do nothing. Filter the preview first to apply only part of it – `subset(m, name != "lb")` – or build the frame by hand with the same columns.

Nothing is written. The set comes back edited, and it is stored only when you pass it to `datom_write_set()`; the call ends by saying so. The write's default commit message then names what was added and repointed.

This is the one difference from syncing files, where each table is written as it syncs. A set is saved in one step, so the edit becomes one version, and you can look at the set, or add to it, before it does.

Every member added is read from its source first, which confirms the version exists and records the project that wrote it. The call stops, before changing anything, when:

- the set has moved since the preview was built – a changed row's member is no longer at `version_from`, or a new row's artifact is already in the set. Build the preview again from the current set;

- a row's kind or project disagrees with the artifact it names;
- a new or changed row names a project with no connection in sources, or the same artifact appears in two such rows;
- sources includes the repo's own project, whose members are outputs.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage. File import needs the optional
# rio package.
if (requireNamespace("git2r", quietly = TRUE) &&
    requireNamespace("rio", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  file.copy(
    system.file("extdata", "dm.csv", package = "datom"),
    file.path(tmp, "repo", "input_files", "dm.csv")
  )

  manifest <- datom_sync_manifest(conn)
  result <- datom_sync(conn, manifest)
  print(result[, c("name", "status", "result")])

  unlink(tmp, recursive = TRUE)
}
```

datom_sync_manifest *Preview What a Sync Will Change*

Description

Looks at the files in the project's input_files/ folder and returns one row per file, saying whether it is new, changed, unchanged since it was last synced, or in a format datom cannot read. Nothing is written. Review the result, drop any rows you do not want, then pass it to [datom_sync\(\)](#).

Usage

```
datom_sync_manifest(conn, path = NULL, pattern = "*", sources = NULL)
```

Arguments

conn	A <code>datom_conn</code> object from <code>datom_get_conn()</code> .
path	Optional path to input files directory. Defaults to <code>input_files/</code> inside the repo. Not accepted on a product repo, which reads no files.
pattern	Glob pattern for file matching. Default <code>"*"</code> . On a product repo it filters source artifact names instead.
sources	On a product repo only, and required there: one <code>datom_conn</code> , or a list of them, for the projects the set's inputs come from. Each connection's project name is what members are matched on. Refused on an ordinary repo.

Details

A file counts as changed when its bytes differ from the file last synced under that name.

Files whose format is outside `datom`'s ingestion allowlist (flat tabular formats only) are flagged `"unsupported_format"` up front, without blocking their allowlisted siblings.

On a product repo (mode: product) it maps the repo's set against source projects instead – see "On a product repo" below.

Value

On an ordinary repo, a data frame with columns: name, file, format, original_file_sha, status (one of "new", "changed", "unchanged", "unsupported_format").

On a product repo, a data frame with columns project, name, kind, version_from (NA for a new artifact), version_to (NA for a member that was not compared) and status (one of "new", "changed", "unchanged", "ambiguous", "not_checked", "excluded"). Versions are full 64-character strings.

On a product repo

A product repo owns one set (named in `.datom/project.yaml`), and this call compares that set, as stored, with what each source project holds now. It reads one manifest per source and the stored set; it writes nothing.

Tables and sets are treated alike: a set a source holds gets a row, and a member that is a set is compared exactly as a table member is, so a set built from other sets syncs the same way.

One row per artifact in the sources whose name matches pattern:

- new – no member points at it (every row, when the set has no version yet);
- changed / unchanged – one member points at it, at an older / the current version;
- ambiguous – two or more members point at it (a live table beside a frozen baseline, say), so neither will move. Move one with `datom_update_members()`, narrowing by member and tags.

Plus one row for each member the call did not compare:

- excluded – its artifact is in a source but does not match pattern;
- not_checked – its project was not passed in sources.

The preview never proposes removing a member. A member whose artifact is no longer listed in its source is named in the messages and left pinned. Members in the repo's own project are outputs and get no row: re-derive them, then move them with `datom_update_members()`.

It stops when sources includes the repo's own project, and when a source connection's project name differs from the name that project's own manifest records.

Tables and sets are saved at different points. On an ordinary repo, `datom_sync()` writes each table as it syncs. On a product repo it hands the edited set back, and the set is saved only by `datom_write_set()` – one version for the whole edit, which you can look at or add to first.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # Drop a source file into the repo's input_files/ directory.
  file.copy(
    system.file("extdata", "dm.csv", package = "datom"),
    file.path(tmp, "repo", "input_files", "dm.csv")
  )

  manifest <- datom_sync_manifest(conn)
  print(manifest[, c("name", "format", "status")])

  unlink(tmp, recursive = TRUE)
}
```

`datom_update_members` *Repoint a Set's Members at Newer Versions*

Description

Moves members of a set forward to the versions that are current now, and returns the set with those pointers changed. **Nothing is written:** the report you see is the dry run, and the set is stored only when you pass the result to `datom_write_set()`.

Usage

```
datom_update_members(  
  x,  
  conn,  
  member = NULL,  
  tags = NULL,  
  version_from = NULL,  
  version_to = NULL  
)
```

Arguments

x	A <code>datom_set</code> , from <code>datom_get_set()</code> or <code>datom_assemble_set()</code> .
conn	A <code>datom_conn</code> from <code>datom_get_conn()</code> , or a list of them – one per project the selected members belong to.
member	Optional: the member to repoint, as its name, a member record, or a link. <code>NULL</code> (the default) selects every member.
tags	Optional named list of labels narrowing the selection, e.g. <code>list(type = "input")</code> . A member matches when it carries every label listed.
version_from	Optional version, or a prefix of one, narrowing the selection to the member pinned at it.
version_to	Optional exact version to move to. <code>NULL</code> (the default) means whatever that artifact's project reports as current. Requires the selection to resolve to a single member.

Details

This is the operation a product needs when its inputs move on – a hundred members of which thirty upstream tables have advanced. Doing it by hand is list surgery on what `datom_get_set()` returned, and two of the obvious spellings are silently wrong: filtering members by name drops *every* version of that name, and rebuilding a pointer from its name plus a new version drops that member's labels.

Value

x with the matching members repointed, and what changed appended to its `datom_edits` attribute.

Which connection to pass

One per project the set spans, since access in `datom` is per project. A set whose members all live in the project that owns it needs only that one connection; a product drawing on three studies needs three, in a list.

A set's projects can be listed **offline, with no connection at all**:

```
unique(datom_list_members(x)$project)
```

Which members move

With no member, **every** member – refreshing everything is the common case and rerunning it changes nothing. Otherwise select one the way `datom_fetch_member()` does: by name, by a member record, or by a link. tags and version_from narrow either a name or the sweep, so tags = list(release = "live") repoints the labelled members and leaves the rest pinned.

A name that matches more than one member **aborts**, because the request cannot be honoured as typed; a *sweep* that meets the same pair skips it and says so, because refusing a whole refresh over one frozen baseline would make the first update on such a set an error.

Which version each member moves to

version_to omitted, each selected member moves to the version its own project reports as current, and every move is reported before anything is written. That is the one place datom infers "newest", and the reason it is allowed here is in the verb's name: a pointer *constructor* requires an explicit version (`datom_member()` refuses to guess), while a verb whose whole meaning is *move this forward* states the time-dependence up front and then says what it picked. What the set records is still an exact version, so a script that later reads that set is as reproducible as ever.

version_to supplied, the selected member moves to exactly that version – a rollback to a known-good, or a deliberate step to something that is not the newest. It requires the selection to resolve to one member, because one explicit version across several artifacts is not a meaning. It is also what makes this verb better than removing and re-adding a member: **the labels come with it**, where re-adding makes you retype them.

What it declines to do, and how

Situation	Response
a member's project has no supplied connection	the whole call is refused, naming that project
a member's artifact no longer appears in its project	reported, and its pin is left alone
two members share a name and a project	both skipped and reported

The first refuses because whether those members moved is unknowable, and reporting them as unchanged would state something nothing checked. The second does not, because the answer *is* known: the pinned version is immutable and still reads, so the set stays writable.

What comes back

The set it was handed, with matching members repointed and each moved member's labels byte-identical to what they were.

When something moved, a `datom_set`'s version and data_sha are emptied: they described the payload it was read as, and that is no longer what the object holds. When nothing moved they are left alone, because the object still describes exactly that stored version.

The returned object also carries a log of what changed, which `datom_write_set()` uses for the commit message when you pass no message of your own – so git log names what moved instead of saying Update {name}. `datom_remove_members()` adds to the same log, so editing both ways before you write produces one message describing both. Passing x\$members rather than x to the write loses that and nothing else.

See Also

`datum_remove_members()` to drop members instead, `datum_write_set()` to store the result, `datum_list_members()` to see what a set holds, `datum_member()` to build a pointer from scratch.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datum-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datum_store(
    data = datum_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datum_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datum_get_conn(file.path(tmp, "repo"), store)

  dm <- datum_example_data("dm")
  datum_write(conn, data = dm, name = "dm")
  datum_write_set(conn, list(
    datum_member(conn, "dm", datum_history(conn, "dm")$version[1],
      tags = list(type = "input"))
  ))

  # The table moves on, so the set now cites an older version of it.
  datum_write(conn, data = dm[-1, , drop = FALSE], name = "dm")

  x <- datum_get_set(conn, "example_product")
  x <- datum_update_members(x, conn)

  # The label came with it, and nothing is stored until the write.
  print(datum_list_members(x))
  datum_write_set(conn, x)

  unlink(tmp, recursive = TRUE)
}
```

Description

Checks that git metadata matches S3 storage for all tables and repo-level files. Reports mismatches as a structured result.

Usage

```
datum_validate(conn, fix = FALSE)
```

Arguments

conn	A datum_conn object from <code>datum_get_conn()</code> .
fix	If TRUE, attempts to fix inconsistencies by syncing data-side metadata (manifest + per-artifact metadata) to storage, and by restoring a set's payload when storage has lost it – git holds {name}/set.json, so those bytes are recoverable. A restore happens only when the stored object is absent and only when the clone's bytes hash to the document_sha already recorded; a stored payload is never overwritten and its recorded hash is never recomputed. A missing table payload (data_missing_s3) cannot be repaired: the parquet bytes are never in the clone. Those tables are named in a warning and need <code>datum_write()</code> re-run with the source data.

Value

A list with:

valid Logical — TRUE if everything is consistent.

repo_files Data frame of repo-level file checks.

tables Data frame of per-artifact checks, one row per artifact of either kind, with a kind column. Named tables for compatibility.

fixed Logical — TRUE if fix = TRUE was applied.

What is checked per artifact

Both kinds of artifact are checked, and the payload check branches on kind: a table's payload is a parquet object, a set's is a JSON document at {name}/{data_sha}.json. A **set** is checked further, because a payload whose members have gone is a citation that no longer resolves:

- every member's pinned version must still exist in this project's storage. **One level deep only** – a member that is itself a set is confirmed to exist and its own member list is never opened, so the cost of validating a set never depends on the tree beneath it. Validating an inner set is a separate call against that set's own project.
- a member recorded as belonging to **another project** is checked as a well-formed pointer only. This connection sees one namespace, so an existence check there would report every cross-project member as rotten.
- the set must record the hash of its stored payload, without which no reader can verify it.

Statuses reported in the tables frame: metadata_missing_s3, history_missing_s3, data_missing_s3, members_unresolvable, document_sha_missing, and kind_unsupported for an artifact whose metadata declares a kind this version of datom does not know – reported rather than fatal, with that row’s payload left unchecked.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  datom_validate(conn)

  unlink(tmp, recursive = TRUE)
}
```

datom_write

Save a Data Frame as a datom Table

Description

Saves a data frame as a new version of a named [table](#): the data goes to storage, and a record of the change is committed and pushed to the project’s GitHub repository. If nothing has changed since the last version, nothing is saved. To bring in files rather than data frames, use [datom_sync\(\)](#).

Usage

```
datom_write(
  conn,
  data = NULL,
  name = NULL,
  metadata = NULL,
  message = NULL,
  parents = NULL,
  .source_lineage = NULL,
```

```

    .table_type = "derived",
    .original_file_sha = NULL,
    .original_format = NULL
  )

```

Arguments

<code>conn</code>	A <code>datom_conn</code> object from <code>datom_get_conn()</code> .
<code>data</code>	Data frame to write. If <code>NULL</code> with <code>name</code> , does metadata-only sync.
<code>name</code>	Table name. If <code>NULL</code> with <code>NULL</code> data, mirrors the clone's storage-side documents for every artifact of either kind: the manifest, and each artifact's metadata, version history and versioned snapshots. On that route a set whose stored payload is missing also has it restored from the clone – see <code>datom_validate()</code> , which shares the mechanism, for the conditions on that.
<code>metadata</code>	Optional list of custom metadata.
<code>message</code>	Optional commit message.
<code>parents</code>	Optional list of parent records produced by <code>datom_parent()</code> , each carrying source, table, version, data_sha, and source_lineage. When supplied, the table's source_lineage is derived as the deduplicated union of the parents' source_lineage and each parent is recorded lean (source, table, version, data_sha). <code>NULL</code> if no lineage is recorded. There is no public source_lineage parameter; it is always derived from parents.
<code>.source_lineage</code>	Internal. Flat list of transitive non-derived source descriptors (each with project, table, version_sha) for the imported self-entry path, set by <code>datom_sync()</code> . Unused on the derived (parents) path.
<code>.table_type</code>	Internal. "derived" (default) or "imported" (set by <code>datom_sync()</code>).
<code>.original_file_sha</code>	Internal. SHA of source file (set by <code>datom_sync()</code>); <code>NULL</code> for derived.
<code>.original_format</code>	Internal. Original file format (set by <code>datom_sync()</code>); <code>NULL</code> for derived.

Value

List with deployment details.

Examples

```

# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(

```

```

    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # --- Basic write (no lineage) ---
  dm <- datom_example_data("dm")
  datom_write(conn, data = dm, name = "dm")

  # --- Write with a single parent ---
  # Each parent's data_sha and lineage are resolved by datom_parent.
  lb <- datom_example_data("lb")
  datom_write(conn, data = lb, name = "lb")
  lb_summary <- aggregate(
    list(n = lb$LBTESTCD), by = list(LBTESTCD = lb$LBTESTCD), FUN = length
  )
  datom_write(
    conn,
    data = lb_summary,
    name = "lb_summary",
    message = "Lab test counts",
    parents = list(
      datom_parent(conn, "lb", datom_history(conn, "lb")$version[1])
    )
  )

  # --- Write with multiple parents ---
  # The source lineage is derived as the union of the parents' lineages.
  dm_lb_merged <- merge(dm, lb, by = "USUBJID")
  datom_write(
    conn,
    data = dm_lb_merged,
    name = "dm_lb_merged",
    message = "Demographics joined with lab results",
    parents = list(
      datom_parent(conn, "dm", datom_history(conn, "dm")$version[1]),
      datom_parent(conn, "lb", datom_history(conn, "lb")$version[1])
    )
  )

  print(datom_list(conn))

  unlink(tmp, recursive = TRUE)
}

```

Description

Saves a [set](#): a named list of exact versions of tables (or other sets), plus labels, that you can cite with one version string. A set holds no data, so saving one copies nothing. Saving the same members and labels again creates no new version.

Usage

```
datom_write_set(
  conn,
  members,
  tags = NULL,
  name = NULL,
  message = NULL,
  include_paths = NULL
)
```

Arguments

conn	A <code>datom_conn</code> object from <code>datom_get_conn()</code> , scoped to the product repo (developer role).
members	A list of member records from <code>datom_member()</code> , each pinning one artifact version and optionally carrying its own tags – or a <code>datom_set</code> , from <code>datom_get_set()</code> or <code>datom_assemble_set()</code> , edited or not. Hand-assembled lists are refused.
tags	Optional named list of set-level text labels – facts about the collection itself, such as a description. Same grammar as a member’s tags: a value is one string or several, text only.
name	The set’s name. Defaults to the <code>set:</code> field in <code>.datom/project.yaml</code> ; when supplied it must equal it.
message	Optional commit message. Omitted, it is <code>Update {name}</code> – except for a set edited with <code>datom_add_member()</code> , <code>datom_update_members()</code> or <code>datom_remove_members()</code> , where the default names the members added, moved or dropped, with their versions. So a set assembled in steps gets <code>Update {name}: add N members on its first write</code> . Pass <code>x</code> rather than <code>x\$members</code> to get that, since the change list travels with the object.
include_paths	Optional character vector of repo-relative paths – your own code, <code>renv.lock</code> , build state – staged into the same commit as the set. Never mirrored to storage: the storage namespace holds <code>datom</code> artifacts and nothing else. See the section below.

Details

One repo holds one set. The repo declares which, in `.datom/project.yaml`:

```
mode: product
set: study001-adam
```

Both are checked before anything is hashed or written, so a repo that has not declared itself a product repo is refused with nothing left behind.

Value

Invisibly, a list with name, data_sha, metadata_sha (the version), member_count (the count after normalisation), action ("none" or "full") and commit_sha.

What a set carries, and what it does not

User metadata is **tags**, and there is no metadata = parameter: a description is a tag, and a second channel for the same thing would be two places to look. There is no view or navigation configuration either – a folder-like hierarchy is a projection a consumer computes over tags, and any number of them cost nothing precisely because none is stored.

A set records **no** parents and **no** source_lineage. Members are references, not derivation: lineage flows through tables, and the set is how you *found* a table rather than how data reached it.

Versions, and what moves one

The version covers the **whole payload**, members and tags alike. So editing a tag or a description mints a new version, which is intended: a set exists to be citable, and "same citation, different labels" would be a lie to whoever cited it. What does **not** mint a version is a purely syntactic edit – reordering tag values or members, repeating a label, or writing a single label as a one-element array. Those are normalised on the way in, so re-writing an identical payload is a no-op.

Your code does not move a version either, even though it travels in the same commit (see include_paths below). Refactor your build script, re-run it, get the same members and tags, and nothing is minted: the write is the usual no-op. `datom_history()` then shows the version it showed before, with a commit_sha pointing at the commit that **first** produced that payload – a commit that does not contain the code you just wrote. That is the recorded value doing its job rather than going stale; see `datom_history()` for why the commit is deliberately not part of the version.

Where the payload lives

Two copies, at two deliberately different addresses. Git holds {name}/set.json at one stable path, modified in place, so git carries the history and git diff between two versions shows which members changed. Storage holds the same bytes content-addressed at {name}/{data_sha}.json, so a reader with no clone can fetch an exact version. Any past version is still reconstructible from the clone alone with git show <commit>:{name}/set.json.

Carrying your code and environment into the same commit

include_paths stages paths you name into the **one** commit that carries the payload and its meta-data. So checking out a set version's commit yields the data pointers, the logic that produced them **and** the environment they ran in – one clone, one checkout, the whole product. The joint version is **structural**: nothing records a link between the set and your files, because the commit *is* the link.

```
datom_write_set(conn, members,
                include_paths = c("R", "dp", "renv.lock"))
```

Four refusals, all of them before anything is hashed or written, so a refusal leaves nothing behind: a path that does not exist, a path outside the clone, a path datom owns (.datom/, the set, any artifact directory), and a path .gitignore excludes. The last one matters because git stages an ignored

path silently and without complaint, which would leave the set version claiming a commit that omits exactly the file you named. Refusals win over the no-op below, since they are settled before change detection runs.

An unchanged set is still a no-op, however dirty those paths are. No commit, no version, and a message pointing at `datom_repo_commit()`, which is the verb for committing your own content at a moment you chose. A data write that quietly committed work in progress is the thing datom's explicit file lists exist to prevent, and idempotency must not become a side door into it.

Outputs must be built from the inputs the set pins

A table written with parents records which versions it was derived from. When a set lists such a table and also lists one of its parents, the write checks they agree: if the set pins the parent at a different version from the one the table was built from, and not at that version too, the write stops and names the member, the parent and both versions. Nothing is written. Re-derive the output with `datom_parent(x = )`, which takes the versions from the set, or move the input with `datom_update_members()`.

Only tables in the set's own project are checked, and a parent the set does not list is not checked. A set carrying one table at two versions (a live copy beside a frozen baseline) passes as long as one of them is the version used. Each such member's recorded metadata is read from storage; if it cannot be read – a version that does not exist, or storage that cannot be reached – the write stops too.

Editing a set that already exists

Read it, change it, write it back. `members` accepts a `datom_set` from `datom_get_set()` directly, so the loop needs no unpacking:

```
x <- datom_get_set(conn, "study001-adam")
x$members <- c(x$members, list(datom_member(conn, "lb", v)))
datom_write_set(conn, x)
```

The set's own tags come along with it unless `tags` is supplied, so a read-append-write cannot silently drop the description. Passing `x$members` instead works too, and there `tags` is yours to carry.

A set built in steps with `datom_assemble_set()` and `datom_add_member()` is written the same way, and pipes into the write:

```
x |> datom_write_set(conn = conn)
```

A set is written into its own repo

A `datom_set` records its name and the project it belongs to. Both are checked before anything is hashed or written: a set named for another repo's declared set, or belonging to another project, stops the write. Writing it anyway would move it into this repo's project without saying so – the name check alone would miss that, since two product repos may declare the same set name. A name argument that disagrees with the set's own name stops it too. A plain list of member records carries neither, so neither is checked.

See Also

`datom_member()` to declare a member, `datom_assemble_set()` to build a set a member at a time, `datom_write()` for tables.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)

  store <- datom_store(
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  # A product repo declares itself as one and names the single set it owns.
  datom_init_repo(file.path(tmp, "repo"), "example_project", store,
    mode = "product", set = "example_product")

  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  # A set points at versions that already exist.
  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  datom_write(conn, data = datom_example_data("lb"), name = "lb")

  members <- list(
    datom_member(conn, "dm", datom_history(conn, "dm")$version[1],
      tags = list(type = "input")),
    datom_member(conn, "lb", datom_history(conn, "lb")$version[1],
      tags = list(type = "output", domain = c("safety", "labs")))
  )

  datom_write_set(
    conn, members,
    tags = list(description = "Example product for STUDY-001")
  )

  print(datom_list(conn))

  unlink(tmp, recursive = TRUE)
}
```

Description

Check if Object is a datom Store

Usage

```
is_datom_store(x)
```

Arguments

x Object to test.

Value

TRUE or FALSE.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store(
  data = datom_store_local(path = tmp),
  data_repo_url = "https://github.com/example/my-project",
  validate = FALSE
)
is_datom_store(store)
is_datom_store("not a store")
unlink(tmp, recursive = TRUE)
```

is_datom_store_local *Check if Object is a Local Store Component*

Description

Check if Object is a Local Store Component

Usage

```
is_datom_store_local(x)
```

Arguments

x Object to test.

Value

TRUE or FALSE.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store_local(path = tmp, validate = TRUE)
is_datom_store_local(store)
is_datom_store_local("not a store")
unlink(tmp, recursive = TRUE)
```

is_datom_store_s3	<i>Check if Object is an S3 Store Component</i>
-------------------	---

Description

Check if Object is an S3 Store Component

Usage

```
is_datom_store_s3(x)
```

Arguments

x	Object to test.
---	-----------------

Value

TRUE or FALSE.

Examples

```
s3 <- datom_store_s3(
  bucket = "my-datom-bucket",
  access_key = "AKIAIOSFODNN7EXAMPLE",
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY",
  validate = FALSE
)
is_datom_store_s3(s3)
is_datom_store_s3("not a store")
```

```
is_datom_store_s3_creds
```

Check if Object is a Credentials-Only S3 Store Component

Description

Check if Object is a Credentials-Only S3 Store Component

Usage

```
is_datom_store_s3_creds(x)
```

Arguments

x Object to test.

Value

TRUE or FALSE.

Examples

```
creds <- datom_store_s3_creds(
  access_key = "AKIAIOSFODNN7EXAMPLE",
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
)
is_datom_store_s3_creds(creds)
is_datom_store_s3_creds("not a store")
```

```
is_valid_datom_repo
```

Check if Path is a Valid datom Repository

Description

Validates datom repository structure. Used internally and by dpbuild.

Usage

```
is_valid_datom_repo(
  path,
  checks = c("all", "git", "datom", "renv"),
  verbose = FALSE
)
```

Arguments

path	Path to evaluate.
checks	Which checks to perform. Any combination of "all", "git", "datom", "renv".
verbose	If TRUE, prints which tests passed/failed.

Value

TRUE or FALSE.

Examples

```
# A plain directory is not a valid datom repository.
tmp <- tempfile("datom_valid_")
dir.create(tmp)
is_valid_datom_repo(tmp)
unlink(tmp, recursive = TRUE)
```

print.datom_conn	<i>Print a datom Connection</i>
------------------	---------------------------------

Description

Displays a clean summary without exposing credentials or the S3 client.

Usage

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

Arguments

x	A datom_conn object.
...	Ignored.

Value

Invisible x.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a
# local directory for object storage.
if (requireNamespace("git2r", quietly = TRUE)) {
  tmp <- tempfile("datom-example-")
  remote <- file.path(tmp, "remote.git")
  dir.create(remote, recursive = TRUE)
  git2r::init(remote, bare = TRUE)
```

```

store <- datom_store(
  data = datom_store_local(file.path(tmp, "storage")),
  github_pat = "example-token", # role selector; a local remote needs none
  data_repo_url = remote,
  validate = FALSE
)
datom_init_repo(file.path(tmp, "repo"), "example_project", store)
conn <- datom_get_conn(file.path(tmp, "repo"), store)

print(conn)

unlink(tmp, recursive = TRUE)
}

```

<code>print.datom_link</code>	<i>Print a Member Link</i>
-------------------------------	----------------------------

Description

Print a Member Link

Usage

```

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

```

Arguments

<code>x</code>	A <code>datom_link</code> from a member of a set read with <code>datom_get_set()</code> .
<code>...</code>	Ignored.

Value

Invisible x.

Examples

```

# See datom_get_set() for a runnable set example; a link is one of its
# members' `$$fetch` elements.
print(names(formals(datom_get_set)))

```

print.datom_set	<i>Print a datom Set</i>
-----------------	--------------------------

Description

One line per member – name, kind, and its tags as compact key=value pairs, or – when it has none – plus the route to a member’s content. Long member lists are truncated. A set not yet written shows version NA.

Usage

```
## S3 method for class 'datom_set'
print(x, ..., n = 20L)
```

Arguments

x	A datom_set, from datom_get_set() or datom_assemble_set() .
...	Ignored.
n	Maximum number of members to list.

Details

Tags are open-keyed by design, so there is no fixed column layout to print them in.

Value

Invisible x.

Examples

```
# See datom_get_set() for a runnable example that prints a set.
print(names(formals(datom_get_set)))
```

print.datom_store	<i>Print a datom Store</i>
-------------------	----------------------------

Description

Displays store configuration with masked secrets.

Usage

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

Arguments

x	A datom_store object.
...	Ignored.

Value

Invisible x.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store(
  data = datom_store_local(path = tmp),
  data_repo_url = "https://github.com/example/my-project",
  validate = FALSE
)
print(store)
unlink(tmp, recursive = TRUE)
```

```
print.datom_store_local
```

Print a Local Store Component

Description

Displays store configuration.

Usage

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

Arguments

x	A datom_store_local object.
...	Ignored.

Value

Invisible x.

Examples

```
tmp <- tempfile("datom_store_")
store <- datom_store_local(path = tmp, validate = TRUE)
print(store)
unlink(tmp, recursive = TRUE)
```

```
print.datom_store_s3
```

Print an S3 Store Component

Description

Displays store configuration with masked secrets.

Usage

```
## S3 method for class 'datom_store_s3'  
print(x, ...)
```

Arguments

x	A datom_store_s3 object.
...	Ignored.

Value

Invisible x.

Examples

```
s3 <- datom_store_s3(  
  bucket = "my-datom-bucket",  
  access_key = "AKIAIOSFODNN7EXAMPLE",  
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY",  
  validate = FALSE  
)  
print(s3)
```

```
print.datom_store_s3_creds
```

Print a Credentials-Only S3 Store Component

Description

Displays masked credentials and a note that location is resolved from ref.json at connection time.

Usage

```
## S3 method for class 'datom_store_s3_creds'  
print(x, ...)
```

Arguments

x	A datom_store_s3_creds object.
...	Ignored.

Value

Invisible x.

Examples

```
creds <- datom_store_s3_creds(  
  access_key = "AKIAIOSFODNN7EXAMPLE",  
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY"  
)  
print(creds)
```

```
print.datom_summary     Print a datom_summary
```

Description

Print a datom_summary

Usage

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

Arguments

x	A datom_summary object.
...	Ignored.

Value

Invisible x.

Examples

```
# Offline, self-contained: a bare git repo stands in for GitHub and a  
# local directory for object storage.  
if (requireNamespace("git2r", quietly = TRUE)) {  
  tmp <- tempfile("datom-example-")  
  remote <- file.path(tmp, "remote.git")  
  dir.create(remote, recursive = TRUE)  
  git2r::init(remote, bare = TRUE)  
  
  store <- datom_store(  

```

```
    data = datom_store_local(file.path(tmp, "storage")),
    github_pat = "example-token", # role selector; a local remote needs none
    data_repo_url = remote,
    validate = FALSE
  )
  datom_init_repo(file.path(tmp, "repo"), "example_project", store)
  conn <- datom_get_conn(file.path(tmp, "repo"), store)

  datom_write(conn, data = datom_example_data("dm"), name = "dm")
  print(datom_summary(conn))

  unlink(tmp, recursive = TRUE)
}
```

Index

connection, [15](#)

datom (datom-package), [3](#)

datom-package, [3](#)

datom_add_member, [4](#)

datom_add_member(), [7](#), [8](#), [72](#), [74](#)

datom_assemble_set, [7](#)

datom_assemble_set(), [4–6](#), [36](#), [61](#), [65](#), [72](#), [74](#), [75](#), [81](#)

datom_check_hashable, [9](#)

datom_clone, [10](#)

datom_clone(), [23](#)

datom_example_cutoffs, [11](#)

datom_example_cutoffs(), [12](#)

datom_example_data, [12](#)

datom_fetch_member, [13](#)

datom_fetch_member(), [19](#), [28](#), [31](#), [58](#), [66](#)

datom_get_conn, [15](#)

datom_get_conn(), [4](#), [5](#), [7](#), [13](#), [16](#), [18](#), [19](#), [22](#), [26](#), [29](#), [31](#), [33](#), [34](#), [45](#), [53](#), [59](#), [60](#), [63](#), [65](#), [68](#), [70](#), [72](#)

datom_get_lineage, [16](#)

datom_get_lineage(), [4](#), [18](#)

datom_get_parents, [18](#)

datom_get_parents(), [17](#)

datom_get_set, [19](#)

datom_get_set(), [4–7](#), [13](#), [21](#), [27](#), [31](#), [34](#), [36](#), [57](#), [61](#), [65](#), [72](#), [74](#), [80](#), [81](#)

datom_history, [21](#)

datom_history(), [4](#), [6](#), [29](#), [34](#), [50](#), [51](#), [73](#)

datom_init_repo, [23](#)

datom_init_repo(), [3](#), [53](#)

datom_lineage_union, [25](#)

datom_list, [26](#)

datom_list(), [50](#), [51](#)

datom_list_members, [27](#)

datom_list_members(), [7](#), [14](#), [37](#), [58](#), [67](#)

datom_member, [28](#)

datom_member(), [4–8](#), [20](#), [66](#), [67](#), [72](#), [75](#)

datom_parent, [30](#)

datom_parent(), [4](#), [18](#), [29](#), [30](#), [70](#)

datom_projects, [32](#)

datom_pull, [33](#)

datom_pull(), [43](#)

datom_read, [34](#)

datom_read(), [4](#), [13](#), [20–22](#), [50](#), [51](#)

datom_remove_members, [35](#)

datom_remove_members(), [6](#), [66](#), [67](#), [72](#)

datom_repo_attach_governance, [38](#)

datom_repo_attach_governance(), [50](#)

datom_repo_commit, [39](#)

datom_repo_commit(), [43](#), [74](#)

datom_repo_delete, [41](#)

datom_repo_delete(), [38](#), [44](#), [48](#)

datom_repo_push, [42](#)

datom_repo_push(), [40](#)

datom_repo_set_data_store, [44](#)

datom_repo_set_data_store(), [38](#), [42](#), [47](#)

datom_status, [45](#)

datom_storage_copy, [46](#)

datom_storage_copy(), [44](#), [52](#)

datom_storage_delete_prefix, [48](#)

datom_storage_delete_prefix(), [41](#), [42](#), [47](#)

datom_storage_list, [49](#)

datom_storage_list(), [47](#), [51](#), [52](#)

datom_storage_read_json, [50](#)

datom_storage_verify, [51](#)

datom_storage_verify(), [44](#), [47](#)

datom_store, [53](#)

datom_store(), [3](#)

datom_store_local, [55](#)

datom_store_local(), [53](#)

datom_store_s3, [55](#)

datom_store_s3(), [53](#)

datom_store_s3_creds, [56](#)

datom_structure_members, [57](#)

datom_structure_members(), [5](#), [14](#), [28](#)

datom_summary, [59](#)

`datum_summary()`, 50
`datum_sync`, 60
`datum_sync()`, 4, 62, 64, 69, 70
`datum_sync_manifest`, 62
`datum_sync_manifest()`, 4, 60, 61
`datum_update_members`, 64
`datum_update_members()`, 6, 7, 36, 37, 63, 64, 72, 74
`datum_validate`, 67
`datum_validate()`, 20, 40, 70
`datum_write`, 69
`datum_write()`, 4, 9, 18, 30, 31, 40, 50, 60, 68, 75
`datum_write_set`, 71
`datum_write_set()`, 6–8, 20, 35–37, 61, 64, 66, 67
`developer`, 53

`git2r::clone()`, 10

`is_datom_store`, 75
`is_datom_store_local`, 76
`is_datom_store_s3`, 77
`is_datom_store_s3_creds`, 78
`is_valid_datom_repo`, 78

`lineage`, 30

`member`, 13

`print.datom_conn`, 79
`print.datom_link`, 80
`print.datom_set`, 81
`print.datom_store`, 81
`print.datom_store_local`, 82
`print.datom_store_s3`, 83
`print.datom_store_s3_creds`, 83
`print.datom_summary`, 84
`project`, 23

`set`, 19, 72

`table`, 69