Package 'datom'

September 8, 2026

Title A Unified Framework for Versioned, Traceable Tabular Data

Version 0.1.2

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

RoxygenNote 7.3.3

VignetteBuilder knitr

NeedsCompilation no

Author Afshin Mashadi-Hossein [aut, cre, cph]

Maintainer Afshin Mashadi-Hossein <amashadihossein@gmail.com>

Repository CRAN

Date/Publication 2026-09-08 04:30:02 UTC

Contents

`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

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

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 *Load Example EDC Data*

Description

Loads bundled clinical trial example data for use in examples and vignettes. The data simulates a Phase II study (STUDY-001) with 48 subjects enrolled over 6 months across four SDTM-flavored domains.

Usage

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

Arguments

domain	One of "dm" (demographics, 48 rows), "ex" (exposure, 48 rows), "lb" (labs, ~720 rows: 3 visits x 5 tests per subject), or "ae" (adverse events, ~80 rows).
cutoff_date	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).

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_get_conn *Get a datom Connection*

Description

Flexible connection for both developers and readers.

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` *Get Lineage for a Table*

Description

Reads lineage metadata for a table. Depending on `depth`, returns either the pre-computed transitive source list (`source_lineage`) or the immediate parent list (`parents`). Both fields are stored flat in the table's metadata – no walking or cross-project resolution is performed.

Usage

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

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.
`depth`	One of `"source"` (default) or `"parents"`.

Details

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` *Get Parent Lineage for a Table*

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)
}
```

---

datom_history *Show Version History*

Description

Shows version history for a table by reading version_history.json from S3. Returns the most recent n versions.

Usage

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

Arguments

conn	A datom_conn object from datom_get_conn().
name	Table name.
n	Maximum number of versions to return. Default 10.
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 columns: version, data_sha, timestamp, author, commit_message.

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)
}
```

---

datom_init_repo *Initialize a datom Repository*

Description

One-time setup for data developers. Creates folder structure, initializes git with remote, sets up configuration files, and pushes to S3.

Usage

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

Arguments

path	Path to the project folder. Defaults to current directory.
project_name	Project name, used for S3 namespace and git repo.
store	A datom_store object (from datom_store()). Must have role "developer" (i.e., github_pat provided).
create_repo	If TRUE, create a GitHub repo via API. Mutually exclusive with providing data_repo_url on the store.
repo_name	GitHub repo name when create_repo = TRUE. Defaults to project_name. 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).
git_ignore	Character vector of patterns to add to .gitignore.
.force	If TRUE, skip the S3 namespace safety check. Use only for intentional takeover of an existing S3 namespace. Default FALSE.

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_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
  )

  datom_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)
}
```

`datom_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

```
datom_lineage_union(lineages)
```

Arguments

`lineages`	A list of `source_lineage` lists (each itself a list of entries with `project`, `table`, `version_sha`). `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 <- datom_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) {
  datom_get_lineage(conns[[p$source]], p$table, version = p$version,
                    depth = "source")
})

recomputed <- datom_lineage_union(parent_sls)
recorded   <- datom_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"))
datom_lineage_union(list(sl1, sl2))
```

---

datom_list *List Available Tables*

Description

Lists tables from S3 manifest. Reads `.metadata/manifest.json` from S3 and returns a data frame with one row per table.

Usage

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

Arguments

`conn`	A `datom_conn` object from `datom_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 table info (name, 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_parent` *Declare a parent for lineage*

Description

Resolves a parent table against a single project connection and returns a pure-data lineage record. The parent's authoritative `data_sha` and its `source_lineage` are read from the parent's own versioned metadata snapshot at `{table}/.metadata/{version}.json`; a caller cannot supply or override `data_sha` (there is no `data_sha` parameter). The returned record retains no live connection and is serializable as plain data.

Usage

```
datom_parent(conn, table, version)
```

Arguments

`conn`	A `datom_conn` scoped to the parent's project store, from `datom_get_conn()`.
`table`	Parent table name (single non-empty validated string).
`version`	Parent version (metadata_sha; single non-empty string).

Details

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

A list with exactly `source`, `table`, `version`, `data_sha`, and `source_lineage`. `source` is the parent connection's `project_name`; `source_lineage` is `NULL` when the snapshot carries none.

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

Unified read function with dispatch via `dispatch.json`. Reads from S3 metadata cache for data readers.

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`	Optional context for dispatch (e.g., "default", "cached").
`...`	Additional parameters forwarded to routed function.

Value

Data frame or routed function 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
  )
  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_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 (`datomanager::gov_attach()`).

Usage

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

Arguments

`conn`	A `datom_conn` object with `role = "developer"` and a local data clone (`conn$path`).
`gov_repo_url`	HTTPS clone URL of the governance git repository to record.
`gov_store`	A `datom_store_s3` or `datom_store_local` component for the governance storage. Only its location fields are persisted; credentials are discarded.
`message`	Optional commit message. Defaults to `"Attach governance: {project_name}"`.

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):

- `.datom/governance.json` in the local data clone (git canonical), committed and pushed to the data repo.
- `{prefix}/datom/.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 datom 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

datom_repo_delete(), datom_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_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

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

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_set_data_store` *Rewrite the Data Store Pointer in project.yaml*

Description

Updates `storage.data` in `.datom/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

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

Arguments

`conn`	A `datom_conn` object with `role = "developer"` and a local repo path (`conn$path`).
`new_store`	A `datom_store_s3` or `datom_store_local` component (i.e. the data-side component of a `datom_store()` object, not the full composite).
`message`	Optional commit message. Defaults to `"Update data store: {project_name}"`.

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 `datom_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

datom_storage_copy(), datom_storage_verify(), datom_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)

  # Repoint 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 *Copy All Objects Between Two datom Storage Namespaces*

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)
}
```

`datom_storage_delete_prefix` *Delete All Objects Under a datom Storage Prefix*

Description

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

Usage

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

Arguments

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

Details

Irreversible. Intended for package developers building tools on top of datom (e.g. datomanager for rollback or source deletion after migration). End users performing a full project teardown should use `datom_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("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")

  # 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)
}
```

`datom_storage_list` *List All Objects in a datom Storage Namespace*

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

conn	A datom_conn 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 <- 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_storage_list(conn))

  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 `datom_conn` object (source / reference).
`to_conn`	A `datom_conn` object (destination to verify).
`keys`	Character vector of relative keys (after `{prefix}/datom/`) to verify. `NULL` (default) verifies every key returned by `datom_storage_list(from_conn)`. 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

Bundles a governance store component, a data store component, and git config into a single store object. Role (developer vs reader) is derived from `github_pat` presence.

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

`governance`	A store component (e.g., `datom_store_s3()`) for governance files (dispatch, ref, migration history), or `NULL` for a no-governance store. A no-governance store represents a project that has not yet been promoted to governance (via the datomanager package); `gov_repo_url` and `gov_local_path` must also be `NULL` in that case.
`data`	A store component (e.g., `datom_store_s3()`) for data files (manifest, tables, metadata).
`github_pat`	GitHub personal access token. If provided, role is `"developer"`. If NULL, role is `"reader"`.
`data_repo_url`	GitHub remote URL for the data repository. Required when `github_pat` is provided and `create_repo = FALSE` in `datom_init_repo()`.
`gov_repo_url`	GitHub remote URL for the shared governance repository. The governance repo is created once per org (via the datomanager package) and referenced here by every project that uses it.
`gov_local_path`	Local directory path for the governance clone. If NULL (default), the clone is placed as a sibling of the data repo, named after the basename of `gov_repo_url` (e.g., `"acme-gov"`).
`github_org`	GitHub organization for repo creation. NULL for personal repos.
`github_api_url`	GitHub API base URL. `NULL` (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.

validate	If TRUE (default), validate GitHub PAT via API. Set to FALSE 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` *Create a Credentials-Only S3 Store Component*

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_summary` *Summarize a datom Project*

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`, `total_versions`, `last_updated`, `remote_url`. `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 *Sync Files to datom Repository*

Description

Processes new/changed files from a manifest produced by datom_sync_manifest(). Imports each file via rio::import(), converts to a data frame, and calls datom_write() to store as parquet in S3 with git metadata. Updates the local .datom/manifest.json after each successful write.

Usage

```
datom_sync(conn, manifest, continue_on_error = TRUE)
```

Arguments

conn	A datom_conn object from datom_get_conn().
manifest	Data frame from datom_sync_manifest(), with columns name, file, format, original_file_sha, status.
continue_on_error	If TRUE (default), continues processing remaining tables when one fails. If FALSE, stops on first error. 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.

Value

The manifest data frame augmented with result and error columns. result is "success", "skipped", or "error".

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` *Scan and Prepare Manifest for Sync*

Description

Scans a flat `input_files/` directory and computes file SHAs. Compares against the current `.datom/manifest.json` to detect new or changed files. Returns a manifest data frame for review before calling `datom_sync()`.

Usage

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

Arguments

`conn`	A `datom_conn` object from `datom_get_conn()`.
`path`	Optional path to input files directory. Defaults to `input_files/` inside the repo.
`pattern`	Glob pattern for file matching. Default `"*"`. Files whose format is outside datom's ingestion allowlist (flat tabular formats only) are flagged `"unsupported_format"` up front, without blocking their allowlisted siblings.

Value

Data frame with columns: name, file, format, original_file_sha, status (one of `"new"`, `"changed"`, `"unchanged"`, `"unsupported_format"`).

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_validate` *Validate Git-Storage Consistency*

Description

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

Usage

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

Arguments

`conn`	A `datom_conn` object from `datom_get_conn()`.
`fix`	If `TRUE`, attempts to fix inconsistencies by syncing data-side metadata (manifest + per-table metadata) to storage.

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-table checks.

fixed Logical — `TRUE` if `fix = TRUE` was applied.

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` *Write a datom Table*

Description

Writes data to a datom repository. Commits to git, pushes, and syncs to S3.

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

conn	A datom_conn object from datom_get_conn().
data	Data frame to write. If NULL with name, does metadata-only sync.
name	Table name. If NULL with NULL data, does a data-only metadata sync to storage (manifest + per-table metadata).
metadata	Optional list of custom metadata.
message	Optional commit message.
parents	Optional list of parent records produced by datom_parent(), 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). NULL if no lineage is recorded. There is no public source_lineage parameter; it is always derived from parents.
.source_lineage	Internal. Flat list of transitive non-derived source descriptors (each with project, table, version_sha) for the imported self-entry path, set by datom_sync(). Unused on the derived (parents) path.
.table_type	Internal. "derived" (default) or "imported" (set by datom_sync()).
.original_file_sha	Internal. SHA of source file (set by datom_sync()); NULL for derived.
.original_format	Internal. Original file format (set by datom_sync()); NULL 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)
}
```

`is_datom_store` *Check if Object is a datom Store*

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 *Check if Object is an S3 Store Component*

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` *Print a datom Connection*

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)
}
```

print.datom_store *Print a datom Store*

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