Package: projectData (via r-universe)

November 11, 2026

Title Deferred Access to Installed Data Collections

Version 0.1.0

Description Registers installed, documented data packages as named collections without reading their data, exposes objects through deferred bindings, resolves name conflicts through explicit primary and preference choices, keeps provenance, and offers a loader interface for resources that need format-specific restoration.

License MIT + file LICENSE

Depends R (>= 4.1.0)

Imports utils, stats

Suggests knitr, rmarkdown, testthat (>= 3.2.0), tools, withr (>= 2.5.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

Encoding UTF-8

URL https://github.com/blaserlab/projectData

BugReports https://github.com/blaserlab/projectData/issues

NeedsCompilation no

Author Brad Blaser [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-3168-5423>)

Maintainer Brad Blaser <bradley.blaser@osumc.edu>

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

Date/Publication 2026-11-11 22:51:03 UTC

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

RemoteRef HEAD

RemoteSha dee5e82581aa99a318e8eb82f2b1b31d09879389

Contents

data_attach 2
data_catalog 5
data_collections 7
data_config 7
data_detach 8
data_get 9
data_inspect 10
data_loader 11
data_loader_adapter 13
data_loader_add 14
data_manifest_draft 15
data_manifest_validate 17
data_manifest_write 18
data_masking 20
data_primary 20
data_query 21
data_retry 22

Index **23**

`data_attach` *Attach data packages*

Description

`data_attach()` is how data become available. Given the names of installed data packages, it inventories each one (reading package metadata only, never data values) and puts its objects, by their own names, in one environment, by default named `"projectData:data"`, placed on the search path at `pos`. The objects are deferred active bindings: each is read the first time you use it, with a message first (see `data_config()`). Nothing is attached unless you call this function. Call it again to attach more packages; they join the same environment. Attachment is all-or-nothing: if any package fails validation (for example a suffix collision), nothing is attached.

Usage

```
data_attach(
  packages,
  lib.loc = NULL,
  id = packages,
  suffix = NULL,
  include = NULL,
  primary = NULL,
  replace = FALSE,
  name = "projectData:data",
  pos = 2L,
  on_mask = c("warn", "error", "ignore")
)
```

Arguments

packages	Names of installed packages to attach.
lib.loc	Library directories searched, in order, for each package.
id	Collection ids (default: the package names).
suffix	NULL or one suffix per package (NA for none). Also exposes each object as <object><suffix>, e.g. cds_main.reference.
include	NULL, or a named list (names are collection ids) of object/resource names to expose. A character vector is accepted for one package.
primary	NULL or the id of the collection whose objects keep bare names when contested. Equivalent to calling data_primary().
replace	Replace the registration of collections that already exist (see "Installation lifecycle and replace").
name	Search-path name of the environment.
pos	Search-path position.
on_mask	One of "warn", "error", "ignore".

Details

Attaching registers each package internally (metadata only) and exposes it. A package that data_get() already registered without attaching is reused, not inventoried again, and any resource it already opened stays cached. You never need to register anything yourself.

An attached package is a *collection* with an id (by default the package name). Use id to attach the same package twice, for example two installed versions from different libraries: distinct ids are the only way to keep two installations side by side.

Value

Invisibly, the names now bound in the environment.

Installation lifecycle and replace

The installed copy is resolved once, here, from lib.loc (default .libPaths()), and that installation (its root, version and an identity made from the stat() of its metadata files, never from a loaded namespace) is recorded with the collection. Later changes to .libPaths() cannot redirect it.

Attaching an id that is already attached or registered from the same package, without lib.loc, suffix, include or replace, does nothing **if the recorded installation is unchanged**. The recorded installation is compared with the files on disk each time: a newer version, a reinstall of the same version, a removal, or a library link that now leads elsewhere all count as changed. A changed installation is an error (class projectData_stale_installation) before anything is touched. It names the collection, the path, the registered and current versions and the command that switches. The same check runs before any resource is opened, so a pending read never reads a replaced installation. A lib.loc that selects a different installation for an existing id is likewise an error (projectData_installation_conflict) unless replace = TRUE; any lib.loc, suffix or include for an existing id needs replace = TRUE.

`replace = TRUE` re-registers the collection under the **same id**. The new registration and the resulting name resolution are built and validated first; only then are they committed, so there is never a second collection or a second search-path entry. The old resource records and their caches are always discarded when the installation changed, even if resource names and descriptors are identical, so every later access, attached or explicit, reads from the new installation. Omitted arguments keep the current collection's settings when the same package is replaced: `lib.loc` defaults to the library the collection was found in (not `.libPaths()`), `suffix` and `include` are kept, and so are the primary designation and preferences. Give an argument to change it (`suffix = NA` removes a suffix). Preferences and the primary must remain valid; if they would not, the replacement fails (class `projectData_replace_failed` added) and the previous registration, bindings, preferences and attachment state are kept. Keeping those records does not restore backing files that were already overwritten.

Replacement changes projectData's registration and bindings only. Objects you already assigned elsewhere are not modified or repaired: after `old_cds <- cds_main; data_attach("pkg", replace = TRUE)`, `old_cds` still holds the old returned object, and it may stop working if the installed files it reads from were overwritten. See the "Lifetime of opened objects" section of `data_detach()`.

Name conflicts

Name resolution is metadata-only. The bare name of an object that exists in several attached collections is bound only if you designated a primary collection (`data_primary()`) or a preference (`data_prefer()`); otherwise it is bound to an error naming the candidates. Attaching still succeeds, and the ambiguity shows in `data_catalog()`. Suffix aliases always resolve and must be unique.

Masking

Objects in `.GlobalEnv` shadow the attached environment (it sits after `.GlobalEnv`) and are never touched. `on_mask` controls what happens when exposed names also exist elsewhere on the search path: `"warn"` (default), `"error"` (checked before anything changes) or `"ignore"`.

Selective attachment and sibling objects

`include` limits which objects are exposed (by object or resource name). Opening one object loads its whole resource, so siblings from the same resource are opened and cached together even if not exposed.

Mapping completeness

The catalog's `origin` column says how a resource was discovered (`index`, `adapter` or `manifest`) and its `mapping` column whether the object names were checked: `recorded` (named by the origin, unchecked), `unknown`, `verified` (opened or inspected, and produced) or `mismatch`. Unknown mappings are never guessed. See `data_catalog()`.

Script datasets (`data/*.R`)

Packages that ship non-lazy data scripts are supported. The script is run only when its resource is first opened, by `utils::data()`, in a fresh environment that holds only the script's outputs. Whether a dataset is a script is decided as `utils::data()` decides it, from the lazy-load map and file

names in data/ (so when x.R and x.rda both exist, the script is used), without opening or running anything. Only for a script is the enclosing lookup chain built, once per opening, and never shared between scripts; serialized, lazy-load and text datasets get a plain output environment. The chain is the fixed set of standard startup packages, in ordinary startup order: stats, graphics, grDevices, utils, datasets, methods, then base. Unqualified calls such as head() or model.matrix() therefore work, and method dispatch is unchanged. The set does not follow options("defaultPackages"), startup profiles or the current search path.

The global workspace, projectData's attached data environment and other attached packages are not on the lookup path. Calls to any other package must be qualified (otherpkg::fn()); otherwise opening fails with "could not find function" and the resource is marked failed (see data_retry()). R CMD INSTALL evaluates scripts in its own session, so an unqualified call can install cleanly and fail only here.

Opening a script does not change the search path. utils::data() itself attaches utils while it runs a script; projectData detaches that attachment again afterwards (also on error) only when utils was not attached beforehand. A preexisting attachment and its position are left alone, and no other search-path entry is restored. Changes a script makes deliberately (library(), attach(), options(), setwd(), ...) are the script's own and are not undone, except that a script calling library(utils) while utils is detached cannot be told apart from data()'s attachment and is cleaned up too. The working directory is the script's directory while it runs and is restored by sys.source().

Nothing is attached by the lookup chain. Exports are exposed lazily, as promises for package::name, so package datasets are not retrieved merely because their names are visible, and functions keep their own environments; standard package namespaces are loaded if needed. Limits: the Autoloads environment is not included; a script that calls library(), attach(), globalenv() or get(envir = globalenv()) can still reach or change the session. This arrangement separates names; it is not a security sandbox.

See Also

data_query() to look at packages without attaching them; data_get() to retrieve one object without attaching.

---

data_catalog *Catalog, info, and help for attached collections*

Description

The catalog is a plain data frame (class projectData_catalog) built purely from recorded metadata and resource state. Building or printing it never opens a resource and never calls get(), class(), dim() or object.size() on data. One row per exposed name (original and alias); a contested name appears once per candidate collection.

Usage

```
data_catalog(collection = NULL)
```

```
data_info(name, collection = NULL)

data_help(name, collection = NULL)
```

Arguments

collection	Collection id(s); NULL means all registered.
name	Original object name or exposed name.

Details

Columns: collection, package, version, library, root, resource (the loadable unit), object, exposed_name, kind (original/alias), binding (bound: the bare name resolves to this row; not_bound: lost to a preference or primary; ambiguous: unresolved conflict; unattached: the collection is registered, for example by data_get(), but not attached; excluded: not selected by include; unmapped: object unknown), loader, loader_status, source, help_topic, title, origin, mapping, state, storage, recorded_class.

origin says where the resource's description came from, and never changes after registration: index (the standard package data index and data/ directory), adapter (a registered loader's discover(); the loader column names it) or manifest (an explicit manifest record). It is about discovery only and says nothing about whether the object names are right.

mapping is whether the object name has been checked: recorded (named by the origin, not checked against what the resource produces), unknown (object names not known), verified (the resource was opened or inspected and produced this name) or mismatch (opened, but the name is absent; retrieval then fails with an error listing the observed names, see data_inspect()). Each recorded name is checked individually, and objects returned but not recorded are not listed until inspection, so a partly matching resource is never wholly verified by one matching name. data_inspect() replaces the names with the observed ones, which are verified, and leaves origin as it was.

state is one of unopened, opening, ready, failed, unavailable-loader. ready means the loader returned; for storage = disk-backed this does not mean the data are in memory. recorded_class is whatever the creator declared; it is **recorded, not observed**.

storage describes the object as it exists. For a ready resource it is the classification of the loader that opened it, recorded at that opening (with the loader id and contract version, internally), so it does not change when that loader is later replaced or removed, and the cached object is not reopened. For any other resource it is the declaration of the loader currently registered for it, or unknown when none is. A failed attempt records nothing; a later successful retry records the loader used for it.

loader_status reports current availability for future opening, not how a cached object was opened. It reports whether the loader is usable, from metadata only: registered (a loader with that id is registered in this session), unresolved (not registered, but a manifest names an adapter package that is installed; the adapter has **not** been loaded or checked, so it may still turn out to be malformed or lack the loader), adapter-error (resolution was attempted on access and failed; the access error says why, and the next access tries again) and unavailable (not registered, and no installed adapter is named). Catalog and query operations never load an adapter to improve this label. Loader readiness is separate from the resource: a registered loader can still fail validation or opening. state is unavailable-loader for unavailable and adapter-error.

source is the path of the resource's file relative to root, and it can be NA. That happens when the package index lists a dataset for which no file was found in data/ (for example, a file removed after

installation), or when a manifest resource declares no `Source`. Such a resource is still listed, with its recorded `object` names and `state` `unopened`, because the catalog never opens anything to check. The problem surfaces only on opening: a missing file raises an error naming the resource. When `data/` holds several files for one dataset name (for example `x.R` and `x.rda`), `source` is the file that `utils::data()` selects and that the built-in reader opens, by one shared rule, so the provenance describes what is read. If the selected file is later removed, opening fails with an error rather than falling back to another file; removing an unselected alternative changes nothing. Lazy-loaded datasets all share `source` `data/Rdata.rdb`, which is a database and not one file per dataset.

`data_info()` returns the catalog rows for a name. `data_help()` explicitly opens the help page of the *original* topic in the source package (aliases lead to the same topic); it is separate from inspection.

Value

A `projectData_catalog` data frame; `data_help()` returns the help object from `utils::help()`.

`data_collections` *List registered collections*

Description

Collections are registered automatically, either by `data_attach()` (which also attaches them) or by `data_get()` with `package` (which does not).

Usage

```
data_collections()
```

Value

A data frame, one row per registered collection, with `attached` telling whether its names are exposed on the search path. No data are touched.

`data_config` *Configure projectData session behavior*

Description

With no arguments, `data_config()` just returns the current settings.

Usage

```
data_config(messages, on_open, ignored_claims)
```

Arguments

messages	Logical (default TRUE). Emit a message the first time a resource is opened, before any data are read. It names the objects, the collection and loader, gives the on-disk size of a single source file when there is one, and notes that the resource is opened from the installed package (projectData never installs packages; what a loader or a data script does while opening is up to that code). Turn it off with data_config(messages = FALSE); the setting lasts for the session.
on_open	Optional function called with a list (collection, resource, package, loader, root) immediately before a loader's open function runs, once per uncached opening attempt. Useful for logging and for counting reads in tests. The resource is already marked as opening, so the callback may use other resources but any access that reaches the resource being opened (directly, through an alias or through a sibling object) is a reentrancy error. An uncaught error in the callback is an opening failure: the reader is not run, the resource is marked failed and data_retry() tries again. Use NULL to remove it.
ignored_claims	How to report adapter claims on standard data resources that the built-in reader keeps (see the adapters vignette): "warn" (default, a projectData_ignored_claim warning), "message" (the same text as a message of that class) or "silent". Ambiguous adapter collisions are errors regardless of this setting.

Value

The current configuration, invisibly.

Examples

```
old <- data_config(messages = FALSE)
data_config(messages = old$messages)
```

---

data_detach *Detach data packages*

Description

Removes collections (default: all) from the attached environment and forgets them, along with any preferences and primary designation that refer to them. This covers collections that are only registered (for example by data_get(package =)) as well: their internal registration and cached references to opened resources are dropped, so the next access registers and opens them afresh. When no attached collection remains, the environment is removed from the search path. Only the environment projectData created is touched: it is removed by its identity, so another search-path entry with the same name is left alone.

Usage

```
data_detach(collections = NULL)
```

Arguments

collections	Registered collection ids; NULL detaches everything.

Value

Invisibly, the detached ids.

Lifetime of opened objects

Detaching and replacing an installed package are different events.

- Detaching (and any registration cleanup) never deletes, moves or closes backing files, and does not modify objects you hold. An object you kept in a variable can therefore keep working after detachment, because the files it reads from are still there. Work directories created for loaders stay in the session temporary directory.
- Detaching does **not** protect objects from the installation changing. Fully materialised in-memory values generally stay usable. A disk-backed object that references files inside an installed package depends on those files remaining available and compatible; if the package is replaced or removed it may stop working, or read different data.
- An object that references an adapter-managed working copy is independent of the installation only if the adapter restored every file reference it needs to the working copy, and only for as long as that copy exists (by default, the session temporary directory).

Which case applies to an object is decided by its loader; see data_loader().

---

data_get *Get data objects explicitly*

Description

Returns objects without exposing them on the search path, so it is the most robust form in scripts and functions, and plays the role for projectData that package::object plays for exported data. It never attaches anything and never changes an attached collection.

Usage

```
data_get(name, collection = NULL, package = NULL, lib.loc = NULL)
```

Arguments

name	Object name(s) or alias(es).
collection	Optional collection id (recycled over name).
package	Optional installed package name (recycled over name).
lib.loc	Library directories searched when package has to be registered; checked against the installation of an existing registration.

Details

Unqualified (`package = NULL`, `collection = NULL`): `name` is resolved exactly as a bare name is among the attached collections, including suffix aliases, and an unresolved ambiguity is an error naming the candidates.

By collection id: with `collection`, `name` is an original object name in that registered collection (it also reaches objects that lost a name conflict).

By package: with `package`, the collection is registered automatically if necessary, from `lib.loc` (default `.libPaths()`), and `name` is an original object name in it. Registration reads package metadata only. Retrieval is independent of primary, preferences and suffixes used for attached access. If the package is already registered (attached or not) it is reused: the installation recorded then stays in force, and `lib.loc` is only checked against it, so a second installed copy is never chosen silently. If several collections come from the package, say which with `collection`; `collection` is also the id given to a newly registered collection (default: the package name).

`data_get()` **opens** the requested resource: it is not a metadata query (see `data_query()` and `data_catalog()` for those). Resources not requested stay unopened, but a resource that produces several objects loads all of them. Give several names to open several objects in one call; a single name returns the object, several return a named list.

Explicit and attached access share one cached resource per collection, whichever happens first. The returned object is the shared reference, **not** an independent copy. Registration created by `data_get()` persists for the session until `data_detach()` forgets it.

Value

The object, or a named list for several names.

Examples

```
# Retrieve one object from an installed package without attaching it
head(data_get("mtcars", package = "datasets"))
```

`data_inspect` *Deep inspection of a resource's contents*

Description

Opens the resource through its loader and replaces its object mapping with the objects actually produced (`mapping = "verified"`; `origin` is unchanged). This is the only way to learn the contents of a resource whose mapping is `unknown`. It is never run implicitly. For ordinary `.rda`/`.RData` datasets the file is loaded into an isolated environment; `data/*.R` scripts are **executed**. The opened resource is cached, so a later access does not read it again.

Usage

```
data_inspect(collection, resource = NULL)
```

Arguments

collection	Collection id.
resource	Resource id(s); NULL means every resource whose mapping is unknown.

Details

The mapping is replaced even when none of the recorded names were produced (for example an index entry example for a file that actually holds sample_metadata) or only some were. Ordinary access (data_get()) stays strict and errors on an absent name, pointing here. Inspection only replaces the mapping after the loader succeeded: a genuine opening failure leaves the mapping untouched (and data_retry() applies). The new mapping is committed through the usual validated catalog update (if validation fails, nothing changes). A newly observed name that collides with another collection is handled by the normal conflict rules: it is ambiguous unless a primary, preference or suffix alias resolves it. Names that disappear from the mapping are no longer bound; names that appear are bound or aliased as for any other object. Help topics are carried over only for names that were already recorded.

Names are unique within a collection. Two distinct resources of one collection cannot produce the same object name, whether the collection is attached or only registered (for example by data_get(package =)). The same rule applies when the collection is assembled and to the complete mapping proposed by an inspection, so an observed name already owned by another resource (for instance one whose mapping is unknown, or a wrongly recorded mapping corrected to such a name) is rejected with a projectData_duplicate_object error naming the collection, the object and the competing resource ids. When several resources are inspected together their mappings are validated as one update, so a coordinated correction that is valid as a whole succeeds even if a single resource alone would collide, and an invalid combination commits none of them. Identical names in *different* collections are unaffected and follow the conflict rules above.

A rejected update is a mapping problem, not a loader failure. Resources opened during the inspection stay ready and cached (so repeating the inspection, or data_get() under their existing names, does not read them again), but the object table, bindings, preferences and attachment state are exactly as before; only a committed mapping changes what the catalog reports. To resolve the collision, correct the recorded mapping (for example in a manifest) so that the resources no longer claim the same name.

Value

The updated catalog rows for the collection, invisibly.

---

data_loader *Create a loader definition (loader contract version 1)*

Description

A loader separates three steps. All three receive resolved local information only, and only open may read data values.

Usage

```
data_loader(
  id,
  open,
  discover = NULL,
  validate = NULL,
  storage = c("memory", "disk-backed", "unknown"),
  needs_work_dir = FALSE
)
```

Arguments

id	Namespaced loader id.
open	Function (descriptor, paths, context) returning a named list.
discover, validate	Optional functions as described above.
storage	"memory", "disk-backed" or "unknown". Reported in the catalog; a ready disk-backed object is not necessarily in memory.
needs_work_dir	Allocate a writable work directory for each open.

Details

- discover(root, context): inspect a collection root and return a list of descriptors (or NULL) without reading data values.
- validate(descriptor, root, context): check a descriptor and its files without opening them. Return TRUE, or signal an error / return a character message to reject it.
- open(descriptor, paths, context): restore the resource and return a **named list** of the objects it produces. paths is a list with root, source (absolute path or NA) and work_dir (a path unique to this open attempt when needs_work_dir = TRUE, otherwise NULL).

A descriptor is a list with fields resource (id, required), loader (id; defaults to the discovering loader), source (path relative to the root, or NA), objects (character vector of object names the resource produces, or NULL when unknown), options (named list), title, help_topic, recorded_class (recorded, not observed; optional), and adapter (list(package=, export=), set by manifests).

context is a list built by one helper for validation and opening, so the shared fields are identical: collection (collection id), package, version, library (the library holding the installed package), root (the installation root, the same path passed to discover and validate) and resource (the resource id). During discovery resource is NA. During a standalone data_manifest_validate() call collection is NA. When authoring (data_manifest_draft(), data_manifest_write(), data_manifest_validate()) root is always laid out like an installed package, so paths relative to it work for data/ and extdata/ alike. For a package source tree, package and version come from its DESCRIPTION, library is NA (it is not installed) and root is a temporary view that exists only during the call; for a bare resource directory package, version and library are NA rather than guessed from the directory name. See "Roots and layouts" in data_manifest_draft(). The only information available solely at opening is the paths argument of open (resolved source and work_dir).

`validate` runs without opening: when a collection is registered (if the loader is already registered) and again before every uncached opening attempt, against the loader selected for that attempt. It must only inspect metadata and file paths; it must not restore the dataset or read backing matrices.

Loader ids must be namespaced (`"pkg::name"`); the `projectData::` prefix is reserved for built-in loaders.

Value

An object of class `projectData_loader`.

Adapter responsibilities for disk-backed objects

projectData never copies data and never deletes backing files; what a returned object depends on is decided by open. An adapter must:

- Resolve every path from `paths$source` or `paths$root`, the installed collection root. Do not store absolute paths in the installation or assume a fixed library location.
- Treat installed files as read-only. If the format needs writable storage, set `needs_work_dir = TRUE` and write only under `paths$work_dir`.
- Keep every file the returned object reads from available for as long as the object can be used. Do not delete or move backing files in open or in finalisers that run while the object is still reachable.
- Know, and state in its own documentation, whether the returned object references the installed files or a working copy. Only a working copy makes the object independent of the installation, and only if the adapter restores all required references (not just the main matrix) to it. It is still only as persistent as the working directory, which by default lives in the session temporary directory.

Consequently, an object referencing installed files is not guaranteed to survive replacement or removal of the installation, even though `data_detach()` leaves those files alone.

See Also

`data_loader_add()`, `data_loader_adapter()`

`data_loader_adapter` *Register loaders from an adapter package*

Description

Loads the adapter's namespace (without attaching it), calls its exported function `export` (which takes no arguments), and registers the returned `data_loader()` or list of loaders. Nothing is installed automatically and no other packages are scanned. The export is called once; its result is fully checked (every element is a valid loader, ids are unique within the declaration, and no id collides with the registry under the `replace` policy) before any loader is registered, so registration either succeeds completely or changes nothing. No discover, validate or open function is called while checking.

Usage

```
data_loader_adapter(package, export = "projectData_loaders", replace = FALSE)
```

Arguments

package	Adapter package name.
export	Name of an exported function returning loader(s).
replace	Passed to data_loader_add().

Value

The registered ids, invisibly.

---

data_loader_add *Register, replace, remove, and list loaders*

Description

Loaders live in the current R session only. Registering a loader never runs it; discovery runs when a collection is attached with data_attach() or data_query(). Registration is atomic: when several loaders are registered together (see data_loader_adapter()) and any one is rejected, the registry is left unchanged.

Usage

```
data_loader_add(loader, replace = FALSE)

data_loader_remove(id)

data_loaders()
```

Arguments

loader	A data_loader() object.
replace	Allow replacing a loader with the same id.
id	Loader id.

Details

Replacing a loader (replace = TRUE) affects resources opened afterwards. Resources that are already ready keep their cached objects and are not reopened or revalidated; the catalog storage of a ready resource keeps describing the loader that opened it, while loader_status reports whether a loader is currently available for future opening. Removing a loader leaves ready objects untouched, and marks unopened resources that need it as unavailable-loader (see loader_status in data_catalog()). Built-in loaders cannot be removed or replaced.

Value

`data_loader_add()` and `data_loader_remove()` return the id invisibly; `data_loaders()` returns a data frame.

`data_manifest_draft` *Draft manifest entries from registered adapters*

Description

Runs the `discover` step of registered loaders against a collection root and returns the descriptors they claim, ready to review, edit and pass to `data_manifest_write()`. Discovery uses metadata only: no resource is opened, no data are read and nothing is written.

Usage

```
data_manifest_draft(root, loaders = NULL, layout = "auto")
```

Arguments

`root`	Collection root to inspect: a package source tree (recommended for authoring), an installed package directory, or a standalone resource directory. See "Roots and layouts".
`loaders`	Optional character vector of registered loader ids to use. Defaults to every registered loader that can discover.
`layout`	One of `"auto"` (default), `"installed"`, `"source"` or `"standalone"`; how `root` is interpreted.

Details

Adapters must already be registered (`data_loader_add()` or `data_loader_adapter()`); this function never loads or registers one. It fails if no registered loader can discover resources, or if a loader named in `loaders` is not registered.

Resources that a manifest would have to decide are left out of the result and reported instead, with a `projectData_draft_conflict` warning and the `conflicts` attribute (a data frame with `resource`, `loader`, `reason`):

- a resource claimed more than once (by several loaders, or twice by one),
- when `loaders` is `NULL`, a claim on a resource of the standard data inventory (`data/`), which the built-in reader keeps unless a manifest says otherwise.

Naming the winning loader in `loaders` resolves both: only those loaders are asked, and a claim on a standard dataset is then treated as a deliberate reassignment and kept.

Descriptors do not record which package supplies the loader. To let projectData load an adapter on first access, set `AdapterPackage` (and optionally `AdapterExport`) yourself, e.g. `d$adapter <- list(package = "myAdapter")`.

Value

A list of descriptors, with a `conflicts` attribute. `Source` paths are always relative to the installed package root, never absolute and never prefixed with `inst/`.

Roots and layouts

`Source` paths in a manifest, and the paths adapter `discover()` and `validate()` callbacks work with, are relative to the **installed** package root, where `data/` and the contents of `inst/` sit side by side (`data/example.rda`, `extdata/counts.tinymat`). `root` can be:

- a **package source tree** (a directory with a `DESCRIPTION`, e.g. `my.datapkg/`), the recommended entry point when authoring. projectData builds a temporary view with the installed shape (links to `data/` and to each top-level entry of `inst/`), hands that view's path to the callbacks as `root`, and removes it afterwards. This needs a file system and R session that can create symbolic links. If a link cannot be created the call fails naming the entry; projectData never falls back to copying data, and existing manifests are left untouched. In that case author from an environment that supports links, or use an installed package or an explicitly prepared installed-layout directory as `root`. Cleanup removes only the links created for the call, never their targets. Neither `data/` nor `inst/` is modified, nothing is installed, and no dataset is loaded or script run: standard datasets are recognized from their file names, so their object names stay unknown. Resolved files must stay inside the package's `data/` and `inst/`, with one authoring exception: a `Source` whose first component is a top-level entry of the source `inst/` may pass through symbolic links that lead outside the tree (e.g. `inst/extdata -> /shared/data/project`), provided the final path exists. The exception holds only for a verified source tree (a `DESCRIPTION` and no installation artifacts such as `Meta/` or `R/<pkg>.rdb`), never for `data/`, installed packages or standalone directories, and `Source` stays package-relative (`extdata/example_resource`); the physical target is never stored. Installed packages keep strict containment: a path resolving outside the installed root is rejected. An accessible symlink does not prove the data will ship: verify that `R CMD build` puts regular files in the tarball for your layout.
- an **installed package** directory (has `Meta/package.rds`), used as is.
- a **standalone resource directory** (neither of the above), used as is. This is the earlier convention of passing `inst/` of a source tree; it still works for resources below `inst/`, but cannot see `data/`.

`layout` overrides detection (`"auto"` picks installed if `Meta/package.rds` exists, source if there is a `DESCRIPTION` with a `Package` field, else standalone). Callbacks receive a `context` as described in `data_loader()` with these values:

	`package`, `version`	`library`	`root`
installed	installed metadata	containing library	package directory
source	from `DESCRIPTION`	`NA` (not installed)	temporary view
standalone	`NA` (not verified)	`NA`	the directory

`collection` and `resource` are `NA` while drafting (`resource` is set for `validate()`). A directory name is never used as a package name.

See Also

data_manifest_write(), data_manifest_validate()

---

data_manifest_validate *Validate a projectData manifest*

Description

A manifest is an optional, versioned DCF file at projectData/manifest.dcf in the installed package (inst/projectData/ in the source). It supplies mappings or loader declarations that cannot be discovered automatically. It contains data only: relative paths and loader references, never code.

Usage

```
data_manifest_validate(
  file,
  root = default_manifest_root(file),
  layout = "auto"
)
```

Arguments

file	Path to a manifest file.
root	Root against which Source paths are checked: an installed package, a package source tree, or a standalone resource directory (see the "Roots and layouts" section of data_manifest_draft()). Defaults to two directories above file, except that for <package source>/inst/projectData/manifest.dcf it is the package source.
layout	One of "auto" (default), "installed", "source" or "standalone"; how root is interpreted.

Details

The first record holds ManifestVersion: 1. Each following record describes one resource with fields Resource (required), Loader (required, namespaced id), Source (path relative to the package root), Objects (comma-separated), Title, HelpTopic, Options (key=value pairs separated by ;), RecordedClass, and optionally AdapterPackage and AdapterExport naming a package export (a function returning a data_loader()) that is loaded by namespace on first access if the loader is not yet registered.

A manifest record replaces every automatically discovered candidate with the same Resource.

Value

A list of validated descriptors, invisibly.

Rules for the whole manifest

Reading, validation and `data_manifest_write()` share these rules:

- a resource is declared once; repeating a `Resource`, whether or not the records agree, is an error (`projectData_manifest_error`);
- distinct resources cannot declare the same object name (`projectData_duplicate_object`, naming the competing resources); the same name in separate collections is unaffected;
- `Options` is a list of `key=value` entries separated by `;`. Whitespace around keys and values is ignored and the first `=` splits key from value, so values may contain `=`. Keys must be non-empty and unique (after trimming) within a resource, values non-empty, and every entry needs `=`. Values cannot contain `;`;
- each descriptor must pass the structural and path checks (namespaced loader, relative `Source` existing inside the root, unique object names).

Scope of validation

Validation is metadata-only: no resource is opened, no dataset script run and no adapter package loaded or installed. If the named loader is registered and has a `validate` function it is called; if it is not registered, only the structural rules above are checked and loader-specific validation is not performed. A manifest that passes cannot fail later because of a duplicate declaration inside it, but validation does not check object names against other discovered resources of the collection (that happens when the collection is assembled), nor that a loader will open the data.

See Also

`data_manifest_draft()` and `data_manifest_write()` to produce a manifest.

`data_manifest_write` *Write a projectData manifest*

Description

Serializes descriptors (see `data_loader()` for their fields, or start from `data_manifest_draft()`) to the manifest format read by `data_manifest_validate()`. The manifest is data only. Nothing is opened and no resource is read.

Usage

```
data_manifest_write(
  descriptors,
  file,
  root = default_manifest_root(file),
  overwrite = FALSE,
  layout = "auto"
)
```

Arguments

descriptors	A list of descriptors (or one descriptor).
file	Where to write. There is no default.
root	Root against which Source paths are checked: a package source tree, an installed package or a standalone resource directory (see "Roots and layouts" in data_manifest_draft()). Defaults to two directories above file, as in data_manifest_validate(), except that for <package source>/inst/projectData/manifest.dcf it is the package source, so both data/ and extdata/ sources resolve. Passing inst/ explicitly keeps the earlier standalone behavior. Source is always written relative to the installed package root.
overwrite	Replace an existing file.
layout	One of "auto" (default), "installed", "source" or "standalone"; how root is interpreted.

Details

Before anything is written, every descriptor is validated exactly as data_manifest_validate() would (relative Source that exists below root, namespaced loader id, unique object names). If the loader is registered and has a validate function, that is called too. A loader that is not registered cannot validate: the descriptor is still checked structurally, the skip is recorded in the result and reported with a projectData_validation_skipped message.

A manifest can hold less than a descriptor can. These are errors, not silent changes:

- title, help_topic and recorded_class are one value per resource; a per-object vector is accepted only if all its values are the same.
- options must be a named list of non-empty, single character values. Keys cannot contain = or ;, values cannot contain ;.
- Object names cannot contain commas. No field may contain a line break or leading or trailing whitespace.

Semantic rules are the ones data_manifest_validate() applies to a file (a resource is declared once, distinct resources do not share an object name, option keys are unique and non-empty), through the same code. The restrictions above are format limits that only the writer needs.

The written text is read back and must reproduce the descriptors exactly before the file is put in place; it is written to a temporary file in the target directory and then renamed, so a failure never leaves a partial or half-replaced manifest.

The file is never written inside an installed package, and an existing file is replaced only with overwrite = TRUE. The conventional location is inst/projectData/manifest.dcf in the package source. Existing manifests are replaced as a whole, not merged.

Value

Invisibly, a data frame with one row per resource and a loader_validation column ("ran", or "skipped: ..." with the reason). The file path is in the file attribute.

See Also

data_manifest_draft(), data_manifest_validate()

---

data_masking *Report names masked by, or shadowing, the attached facade*

Description

Checks only names, never values. Objects in .GlobalEnv take precedence over the facade and are never overwritten.

Usage

```
data_masking()
```

Value

A data frame with name and where (the other search-path entry).

---

data_primary *Primary collection and per-object preferences*

Description

These apply to attached collections (registered collections that data_get() created without attaching them take no part in bare-name resolution). The primary collection is the explicit, user-chosen winner for contested bare names; there is no implicit winner based on registration or attachment order. A preference for a specific object overrides the primary. Neither reads data.

Usage

```
data_primary(id)

data_prefer(name, collection)

data_preferences()
```

Arguments

id	Collection id, or NULL to clear. If missing, the current primary is returned.
name	Object name(s).
collection	Collection id(s) to prefer for name, or NULL to clear.

Value

data_primary() returns the primary id (or NA); the others return their result invisibly / a data frame.

data_query *Look inside data packages without attaching them*

Description

Builds the same inventory and catalog that `data_attach()` would, and returns it, **without changing anything**: no environment is attached, nothing is remembered, and no data values are read. Use it to see what a package contains and which names would collide, including collisions with collections that are already attached. The arguments mean the same as in `data_attach()`.

Usage

```
data_query(
  packages,
  lib.loc = NULL,
  id = packages,
  suffix = NULL,
  include = NULL
)
```

Arguments

`packages`	Names of installed packages to attach.
`lib.loc`	Library directories searched, in order, for each package.
`id`	Collection ids (default: the package names).
`suffix`	`NULL` or one suffix per package (`NA` for none). Also exposes each object as `<object><suffix>`, e.g. `cds_main.reference`.
`include`	`NULL`, or a named list (names are collection ids) of object/resource names to expose. A character vector is accepted for one package.

Details

The `binding` column shows what would happen on attachment, given any attached collections and current preferences. A package that would fail to attach (for example, a suffix collision) fails here with the same error. If an id is already attached, the result shows it as it would look after re-attaching with `replace = TRUE`.

Value

A `projectData_catalog` data frame for the queried packages; see `data_catalog()` for the columns.

Examples

```
# A package that ships data, queried without attaching it
head(data_query("datasets")[, c("collection", "object", "resource")])
```

`data_retry` *Retry a failed resource*

Description

Resets `failed` resources to `unopened` so that the next access tries again. A failure is otherwise sticky, so a broken backing file is not re-read on every access.

Usage

```
data_retry(collection, resource = NULL)
```

Arguments

`collection`	Collection id.
`resource`	Resource id(s); `NULL` means all failed resources.

Value

The reset resource ids, invisibly.

Index

.libPaths(), *3, 4*, *10*

class(), *5*

data_attach, 2
data_attach(), *7*, *14*, *21*
data_catalog, 5
data_catalog(), *4*, *10*, *14*, *21*
data_collections, 7
data_config, 7
data_config(), *2*
data_detach, 8
data_detach(), *4*, *10*, *13*
data_get, 9
data_get(), *3*, *5–7*, *11*, *20*
data_help *(*data_catalog*)*, 5
data_info *(*data_catalog*)*, 5
data_inspect, 10
data_inspect(), *6*
data_loader, 11
data_loader(), *9*, *13, 14*, *16–18*
data_loader_adapter, 13
data_loader_adapter(), *13–15*
data_loader_add, 14
data_loader_add(), *13–15*
data_loader_remove *(*data_loader_add*)*, 14
data_loaders *(*data_loader_add*)*, 14
data_manifest_draft, 15
data_manifest_draft(), *12*, *17–20*
data_manifest_validate, 17
data_manifest_validate(), *12*, *17–20*
data_manifest_write, 18
data_manifest_write(), *12*, *15*, *17, 18*
data_masking, 20
data_prefer *(*data_primary*)*, 20
data_prefer(), *4*
data_preferences *(*data_primary*)*, 20
data_primary, 20
data_primary(), *3, 4*
data_query, 21
data_query(), *5*, *10*, *14*
data_retry, 22
data_retry(), *5*
dim(), *5*

get(), *5*
globalenv(), *5*

object.size(), *5*

sys.source(), *5*

utils::data(), *4, 5*, *7*
utils::help(), *7*