Package ‘zot’

October 25, 2026

Version 0.5.0

Title Access 'Zotero' Libraries

Description Interface to the 'Zotero' reference manager <https://www.zotero.org> through its public web API and local client database. Provides paginated reads, versioned writes, read-only local queries, batch plans and resumable ledgers, bibliographic comparison, metadata mapping, and file attachment.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

URL https://kanameishi.github.io/zot/

Depends R (>= 4.1.0)

Imports data.table, DBI, digest, httr2, jsonlite, RSQLite

Suggests knitr, rmarkdown, testthat (>= 3.2.0)

VignetteBuilder knitr, rmarkdown

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Alejandro Verri Kozlowski [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-8535-1170>)

Maintainer Alejandro Verri Kozlowski <averri@fi.uba.ar>

Repository CRAN

Date/Publication 2026-10-25 20:00:03 UTC

Contents

zotApprove 2
zotCollections 3
zotConfig 4
zotExecute 5
zotGetAll 7

zotIdentity . . . 8
zotItems . . . 9
zotLibraries . . . 10
zotLibraryPath . . . 11
zotListItems . . . 11
zotLocalFiles . . . 13
zotLocalItems . . . 14
zotLocalQuery . . . 15
zotLocalRecords . . . 16
zotPreview . . . 18
zotReadBatch . . . 19
zotRequest . . . 20
zotSameItem . . . 21
zotStatus . . . 22
zotTrash . . . 23
zotVerify . . . 24
zotWriteBatch . . . 26
ztAttach . . . 27
ztAttachPlan . . . 28
ztBlockMoveCreate . . . 30
ztBlockMoveTrash . . . 30
ztCensus . . . 31
ztCheckPayload . . . 32
ztCleanName . . . 33
ztCleanPayload . . . 34
ztCleanTitle . . . 34
ztCollectionTree . . . 35
ztCrossref . . . 36
ztMatches . . . 37
ztRefile . . . 38
ztRetitle . . . 39
ztTemplate . . . 40
ztTwins . . . 41

Index **42**

`zotApprove` *Create an approval record for a preview*

Description

Stores a preview digest, optional notes, and a timestamp in an object used by `zotExecute()`.

Usage

```
zotApprove(preview, notes = "")
```

Arguments

preview	A zotPreview() object.
notes	Free-text metadata associated with the record.

Details

This function does not authenticate a person, evaluate an authorization policy, display the plan, or perform a write. The calling application determines how approval records are created. A preview should not be modified after this record is created: zotExecute() checks the approval against the preview digest and recalculates the digest from the plan before making requests.

Value

A zotApproval containing the preview digest.

See Also

zotPreview() and zotExecute().

Examples

```
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "ABCD1234",
  payload = list(list(title = "A reviewed title"))
)
Preview <- zotPreview(Plan, describe = "Synthetic documentation batch")

Approval <- zotApprove(Preview, notes = "Documentation example")
Approval$digest == Preview$digest
```

---

zotCollections *List the collections of a library with their paths*

Description

Reads every collection and derives each one's path from the root, joined with / exactly as ztCollectionTree() names the collections it returns.

Usage

```
zotCollections(config, library = "user")
```

Arguments

config	A zotConfig() object.
library	"user" or a group id.

Value

A data.table in API order with `collectionKey`, `name`, `parentCollection` (`NA` at the root), `path`, `numItems` and `version`.

See Also

`ztCollectionTree()`, `ztRefile()`, `ztCensus()`.

Examples

```
Collections <- list(
  list(key = "ROOT", version = 3L, meta = list(numItems = 2L),
       data = list(name = "Projects", parentCollection = FALSE)),
  list(key = "LEAF", version = 4L, meta = list(numItems = 1L),
       data = list(name = "2026", parentCollection = "ROOT"))
)
Performer <- function(config, method, path, query, body, version) {
  list(status = 200L, headers = list(`total-results` = "2"),
       body = Collections)
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
zotCollections(Config)
```

`zotConfig` *Configure Zotero API access*

Description

Creates a configuration object for requests to the Zotero Web API. The object stores the user identifier, API key, and an optional request performer and file transport used by tests or alternative transports. Arguments may be passed explicitly or read from environment variables, which supports interactive, scripted, and agent-driven use.

Usage

```
zotConfig(
  userID = Sys.getenv("ZOTERO_LIBRARY_ID"),
  key = Sys.getenv("ZOTERO_API_KEY"),
  performer = NULL,
  fileTransport = NULL
)
```

Arguments

`userID`	Zotero user identifier. The default reads `ZOTERO_LIBRARY_ID`.
`key`	Zotero API key. The default reads `ZOTERO_API_KEY`.
`performer`	Optional function used by `zotRequest()` instead of the package HTTP transport.
`fileTransport`	Optional list of the three file-upload steps (`authorize`, `upload`, `register`) used by `ztAttach()` and by the `attach` action of `zotExecute()` instead of the package's storage transport.

Value

An object of class `zotConfig`.

Examples

```
Config <- zotConfig(userID = "42", key = "example-key")
Config
```

---

zotExecute *Execute a previewed batch with a resumable ledger*

Description

Recalculates the plan's digest and requires it to match both the preview and its approval, refuses a ledger written for a different batch, and skips the entries an earlier run completed. Every row is recorded before and after its effect, so an interrupted batch leaves a ledger that names the row whose outcome is unknown.

Usage

```
zotExecute(config, preview, approval, ledgerPath)
```

Arguments

`config`	A `zotConfig()` object.
`preview`	The `zotPreview()` being executed.
`approval`	Its `zotApprove()` record.
`ledgerPath`	JSON ledger file; reuse the same path to resume.

Details

Patch operations read the current item version, use a conditional write, and compare requested fields with a subsequent API read. Create and attach operations persist the new key as `"created"` before reading it back. Trash operations use the status returned by `zotTrash()`. The approval record is application metadata; this function does not independently determine who authorized the batch.

The digest is recalculated from the plan being executed, so a preview whose rows changed after approval is refused instead of written under the old consent. Before each effect the row is written as `"attempt"` and the ledger is replaced atomically; the result overwrites it immediately afterwards. A resumed batch that meets an `"attempt"` stops with a `zotLedgerError`: that row's effect is unknown, and only the caller can reconcile it against the library before retrying.

If a create read-back fails, its `"created"` entry retains the new key and error. Resuming only reads that key and confirms the payload; it never posts the item again. A transport exception during POST remains an `"attempt"` because its effect is unknown. A server failure or a response without a recognized result also remains an attempt. A rejected create response can be retried.

An attach row first recalculates the MD5 of its local file and refuses a file changed since approval with a `zotApprovalError`, recorded as `"failed"` before any request. It then creates the attachment item like a create row and saves its key as `"created"`. On that key it runs Zotero's upload protocol (authorize, where `"exists"` counts as uploaded; upload; register) through the configuration's file transport, then confirms the item's MD5, filename, content type and parent. A failed upload leaves the entry `"created"` with its error; resuming repeats the upload on the same key, after checking the file again, and never creates a second item.

New ledgers have `schema = 2` and identify existing items as `library:key`; creates retain `create-<row>` for block-move plans. Ledgers without a schema keep their original key-only format when every id selects exactly one plan row. An ambiguous legacy ledger is refused without rewriting it. A legacy failed create without a retained key also requires caller reconciliation: older executors could lose the key after a successful POST.

Value

The ledger as a list (class `zotLedger`).

See Also

`zotPreview()`, `zotApprove()`, and `zotVerify()`.

Examples

```
Store <- new.env(parent = emptyenv())
Store$version <- 1L
Store$title <- "Before"
Performer <- function(config, method, path, query, body, version) {
  if (method == "GET") {
    return(list(
      status = 200L, headers = list(),
      body = list(version = Store$version, data = list(title = Store$title))
    ))
  }
  Store$title <- body$title
```

```
  Store$version <- Store$version + 1L
  list(status = 204L, headers = list(), body = NULL)
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "K1",
  payload = list(list(title = "After"))
)
Preview <- zotPreview(Plan, describe = "Synthetic offline patch")
Approval <- zotApprove(Preview, notes = "Documentation example")
LedgerPath <- tempfile(fileext = ".json")
Ledger <- zotExecute(Config, Preview, Approval, ledgerPath = LedgerPath)
Ledger$items[["user:K1"]]$status
unlink(LedgerPath)
```

`zotGetAll` *Fetch every page of a listing endpoint*

Description

Follows `Total-Results` with `start`/`limit` pagination and returns the concatenated item list.

Usage

```
zotGetAll(config, path, query = list(), limit = 100L)
```

Arguments

`config`	A `zotConfig()` object.
`path`	API path starting at the host root (see `zotLibraryPath()`).
`query`	Named list of query parameters.
`limit`	Page size.

Value

A list of parsed items.

See Also

`zotGet()` and `zotRequest()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  Page <- if (query$start == 0L) {
    list(list(key = "A"), list(key = "B"))
  } else {
    list(list(key = "C"))
  }
  list(status = 200L, headers = list(`total-results` = "3"), body = Page)
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
Items <- zotGetAll(Config, path = "/users/42/items", limit = 2L)
vapply(Items, function(Item) Item$key, character(1L))
```

`zotIdentity` *Map a record to the fields compared by strong identity*

Description

Accepts an item as the Web API returns it (with `key`, `data`, `library` and `links`), the `data` of such an item, a row of `zotLocalRecords()`, or an identity list already in this shape, and returns the same identity list for all of them: one rule for every plane.

Usage

```
zotIdentity(record)
```

Arguments

`record`	A list or one-row data.frame in any of the shapes above.

Details

`year` is the first four-digit year in `date`. `author` is the family name (or single-field name) of the first creator of type `author`; an item without one has `NA`. `hasFile` is `TRUE` for an API item whose `links` carry an attachment, the value of an explicit `hasFile` field, and `NA` when unknown, so that the file guard of `zotSameItem()` only acts on observed facts. `library` is `"user"` or the group id when the record says where it lives, `NA` otherwise.

Value

A list: `itemKey`, `library`, `doi`, `md5`, `title`, `year`, `author`, `hasFile`.

See Also

`zotSameItem()`, `ztMatches()`, `zotLocalRecords()`.

Examples

```
Item <- list(
  key = "K1", library = list(type = "group", id = 555L),
  links = list(attachment = list(href = ". . . ")),
  data = list(title = "A record", date = "March 2026", DOI = "10.1000/x",
              creators = list(list(creatorType = "editor", lastName = "First"),
                              list(creatorType = "author", lastName = "Lovelace")))
)
zotIdentity(Item)
```

`zotItems` *Read items by key*

Description

Fetches the current API record of each key in batches of fifty, the maximum the `itemKey` filter accepts. Keys the API does not return are reported in `missing`; absence is a verdict, not an error.

Usage

```
zotItems(config, keys, library = "user", includeTrashed = FALSE, fields = NULL)
```

Arguments

`config`	A `zotConfig()` object.
`keys`	Character vector of item keys, without duplicates.
`library`	`"user"` or a group id.
`includeTrashed`	Whether items in the Trash are returned.
`fields`	Optional character vector: keep only these `data` fields.

Value

A list: `items` (named by key, in the order of `keys`), `missing` and `version` (the library's `Last-Modified-Version`).

See Also

`zotListItems()`, `zotGet()`, `zotVerify()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  list(status = 200L, headers = list(`last-modified-version` = "9"),
       body = list(list(key = "K1", version = 1L,
                        data = list(title = "One", abstractNote = ". . . "))))
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
```

```
      performer = Performer),
  class = "zotConfig"
)
Read <- zotItems(Config, keys = c("K1", "K2"), fields = "title")
names(Read$items)
Read$missing
```

`zotLibraries` *List the personal library and the accessible groups*

Description

Reads the group memberships of the configured user and the current version of every library. The personal library appears first as `"user"`; each group appears under its numeric identifier, the value accepted by the `library` argument of the other functions.

Usage

```
zotLibraries(config)
```

Arguments

`config`	A `zotConfig()` object.

Details

One request lists the groups and one `limit = 1` request per library reads its `Last-Modified-Version` header. The personal library has no name in the API; its `name` is `NA`.

Value

A data.table with `library`, `type` (`"user"` or `"group"`), `name` and `version`.

See Also

`zotLibraryPath()`, `ztCensus()`, `zotCollections()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  if (grepl("/groups$", path)) {
    return(list(status = 200L, headers = list(`total-results` = "1"),
                body = list(list(id = 555L, data = list(name = "Team")))))
  }
  list(status = 200L, headers = list(`last-modified-version` = "12"),
       body = list())
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
```

```
      performer = Performer),
  class = "zotConfig"
)
zotLibraries(Config)
```

`zotLibraryPath` *Build a Zotero library API path*

Description

Build a Zotero library API path

Usage

```
zotLibraryPath(config, library = "user")
```

Arguments

`config`	A `zotConfig()` object.
`library`	`"user"` for the configured user library or a numeric Zotero group identifier.

Value

The API path prefix for the selected library.

See Also

`zotConfig()` and `zotRequest()`.

Examples

```
Config <- zotConfig(userID = "42", key = "example-key")
zotLibraryPath(Config)
zotLibraryPath(Config, library = "5107760")
```

`zotListItems` *List items with a declared bound*

Description

Bounded, version-stable listing of a library, a collection or an item's children. One page is returned unless `all = TRUE`, which follows the pagination up to `max` results and refuses a larger listing before downloading it. The library version is read from every page and a change between pages is reported as an error, so a listing never mixes two states.

Usage

```
zotListItems(
  config,
  library = "user",
  collectionKey = NULL,
  parentKey = NULL,
  top = FALSE,
  since = NULL,
  itemType = NULL,
  includeTrashed = FALSE,
  versions = FALSE,
  fields = NULL,
  limit = PAGE_LIMIT,
  all = FALSE,
  max = NULL
)
```

Arguments

`config`	A `zotConfig()` object.
`library`	`"user"` or a group id.
`collectionKey`	Restrict to the members of this collection.
`parentKey`	Restrict to the children of this item.
`top`	Top-level items only; it cannot be combined with `parentKey`.
`since`	Only objects modified after this library version.
`itemType`	An API item type filter, for example `"-note"`.
`includeTrashed`	Whether items in the Trash are returned.
`versions`	Return the key-to-version map instead of the records.
`fields`	Optional character vector: keep only these `data` fields.
`limit`	Page size, at most one hundred.
`all`	Follow the pagination to the end of the listing.
`max`	Required with `all`: the largest listing accepted.

Value

A list: `items`, `version`, `total` and `truncated`.

See Also

`zotItems()`, `zotGetAll()`, `ztCensus()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
 Page <- if (query$start == 0L) list(list(key = "A"), list(key = "B")) else list(list(key = "C"))
  list(status = 200L,
       headers = list(`total-results` = "3", `last-modified-version` = "7"),
       body = Page)
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
zotListItems(Config, top = TRUE, limit = 2L)$truncated
zotListItems(Config, top = TRUE, limit = 2L, all = TRUE, max = 10L)$total
```

---

zotLocalFiles *Locate attachment files in the local Zotero storage*

Description

Lists non-trashed attachments with their parent, hash and the local file when it exists: imported files live under `storage/<key>/` beside the database, linked files at their absolute path. Filter by the keys of attachments or of their parent records, or by MD5.

Usage

```
zotLocalFiles(
  keys = NULL,
  md5 = NULL,
  path = file.path(Sys.getenv("HOME"), "Zotero", "zotero.sqlite")
)
```

Arguments

`keys`	Optional character vector of attachment or parent item keys.
`md5`	Optional character vector of file hashes.
`path`	Database file; the storage directory is its sibling `storage`.

Details

The database may lag the Web API and a file may be absent locally even when the server holds it: a `NA` in `file` is an observation about this machine, not about the library. `linkMode` follows the client: 0 imported file, 1 imported URL, 2 linked file, 3 linked URL.

Value

A data.table with `itemKey`, `parentKey`, `groupID`, `contentType`, `linkMode`, `md5` and `file`.

See Also

zotLocalRecords(), ztAttach(), zotSameItem().

Examples

```
Path <- tempfile(fileext = ".sqlite")
Connection <- DBI::dbConnect(RSQLite::SQLite(), Path)
DBI::dbWriteTable(Connection, "libraries", data.frame(libraryID = 1L))
DBI::dbWriteTable(Connection, "groups",
                  data.frame(libraryID = integer(), groupID = integer()))
DBI::dbWriteTable(Connection, "items", data.frame(
  itemID = 1:2, key = c("PARENT", "FILE"), libraryID = 1L
))
DBI::dbWriteTable(Connection, "itemAttachments", data.frame(
  itemID = 2L, parentItemID = 1L, linkMode = 0L,
  contentType = "application/pdf", path = "storage:paper.pdf",
  storageHash = "abc"
))
DBI::dbWriteTable(Connection, "deletedItems", data.frame(itemID = integer()))
DBI::dbDisconnect(Connection)
zotLocalFiles(keys = "PARENT", path = Path)
unlink(Path)
```

---

zotLocalItems *List items from the local database*

Description

Returns key, library, type, title, and dates for non-trashed items as a data.table. See zotLocalQuery() for synchronization and read-only details.

Usage

```
zotLocalItems(
  library = NULL,
  path = file.path(Sys.getenv("HOME"), "Zotero", "zotero.sqlite")
)
```

Arguments

library	NULL for every library, "user" for the personal library, or a numeric group id.
path	Database file.

Value

A data.table with key, groupID (0 for personal), itemType, title, dateAdded and dateModified.

See Also

zotLocalQuery() and ztCensus().

Examples

```
# A temporary database with the tables used by zotLocalItems().
Path <- tempfile(fileext = ".sqlite")
Connection <- DBI::dbConnect(RSQLite::SQLite(), Path)
DBI::dbWriteTable(Connection, "libraries", data.frame(libraryID = 1:2))
DBI::dbWriteTable(Connection, "groups",
                  data.frame(libraryID = 2L, groupID = 555L))
DBI::dbWriteTable(Connection, "itemTypes",
                  data.frame(itemTypeID = 1L, typeName = "journalArticle"))
DBI::dbWriteTable(Connection, "items", data.frame(
  itemID = 1:3, key = c("AAA", "BBB", "CCC"), libraryID = c(1L, 2L, 1L),
  itemTypeID = 1L, dateAdded = "2026-01-01", dateModified = "2026-01-01"
))
DBI::dbWriteTable(Connection, "fieldsCombined",
                  data.frame(fieldID = 110L, fieldName = "title"))
DBI::dbWriteTable(Connection, "itemData",
                  data.frame(itemID = 1:3, fieldID = 110L, valueID = 1:3))
DBI::dbWriteTable(Connection, "itemDataValues", data.frame(
  valueID = 1:3, value = c("Personal article", "Group article", "Trashed article")
))
DBI::dbWriteTable(Connection, "deletedItems", data.frame(itemID = 3L))
DBI::dbDisconnect(Connection)

zotLocalItems(library = "user", path = Path)
zotLocalItems(library = 555, path = Path)
unlink(Path)
```

---

zotLocalQuery *Read-only query over the local Zotero database*

Description

Opens the local `zotero.sqlite` database in read-only mode. Local records can lag the Web API until the desktop client synchronizes, and WAL mode can make recent local changes temporarily unavailable to another reader.

Usage

```
zotLocalQuery(
  sql,
  params = list(),
  path = file.path(Sys.getenv("HOME"), "Zotero", "zotero.sqlite")
)
```

Arguments

sql	One read-only SQL statement.
params	Positional query parameters.
path	Database file (default: the standard Zotero location).

Details

The database connection is opened with SQLite's read-only flag, without issuing any PRAGMA, and closed when the query finishes. zotLocalQuery() does not copy or modify the desktop database. The caller remains responsible for supplying a read-only SQL statement. The default path is ~/Zotero/zotero.sqlite; pass path explicitly for a portable installation or a custom Zotero data directory.

While Zotero is writing, it locks the database. A read waits for the write to finish for up to LOCALWAIT, a package constant of 10000 milliseconds (10 s). If the database is still locked after that wait, the query fails with a zotLocalError naming the wait and the original SQLite error; any other SQLite error propagates unchanged.

Value

Query result as a data.table.

See Also

zotLocalItems() and zotRequest().

Examples

```
Path <- tempfile(fileext = ".sqlite")
Connection <- DBI::dbConnect(RSQLite::SQLite(), Path)
DBI::dbWriteTable(Connection, "items", data.frame(key = c("A", "B")))
DBI::dbDisconnect(Connection)
zotLocalQuery("SELECT key FROM items ORDER BY key", path = Path)
unlink(Path)
```

---

zotLocalRecords *Read bibliographic records with their identity fields from the local database*

Description

Returns, for every non-trashed regular item, the fields that zotSameItem() compares: DOI, title, date and its year, the family name of the first author, and whether the item carries a file. Attachments, notes and annotations are not records and are excluded. See zotLocalQuery() for synchronization and read-only details.

Usage

```
zotLocalRecords(
  library = NULL,
  path = file.path(Sys.getenv("HOME"), "Zotero", "zotero.sqlite")
)
```

Arguments

library	NULL for every library, "user" for the personal library, or a numeric group id.
path	Database file.

Details

author is the family name of the creator of type author with the lowest order; a record without an author has NA there and can only match by DOI or file. hasFile is TRUE when a child attachment is an imported or linked file (linkMode 0, 1 or 2); a linked URL is not a file. year is the first four-digit year found in date, the same rule as zotIdentity().

Value

A data.table with itemKey, groupID (0 for personal), itemType, title, doi, date, year, author and hasFile.

See Also

zotLocalItems(), zotIdentity(), ztMatches().

Examples

```
Path <- tempfile(fileext = ".sqlite")
Connection <- DBI::dbConnect(RSQLite::SQLite(), Path)
DBI::dbWriteTable(Connection, "libraries", data.frame(libraryID = 1L))
DBI::dbWriteTable(Connection, "groups",
                  data.frame(libraryID = integer(), groupID = integer()))
DBI::dbWriteTable(Connection, "itemTypes",
                  data.frame(itemTypeID = 1L, typeName = "journalArticle"))
DBI::dbWriteTable(Connection, "items", data.frame(
  itemID = 1L, key = "AAA", libraryID = 1L, itemTypeID = 1L,
  dateAdded = "2026-01-01", dateModified = "2026-01-01"
))
DBI::dbWriteTable(Connection, "fieldsCombined", data.frame(
  fieldID = c(110L, 26L, 14L), fieldName = c("title", "DOI", "date")
))
DBI::dbWriteTable(Connection, "itemData", data.frame(
  itemID = 1L, fieldID = c(110L, 26L, 14L), valueID = 1:3
))
DBI::dbWriteTable(Connection, "itemDataValues", data.frame(
  valueID = 1:3, value = c("A record", "10.1000/x", "2026-03-01")
))
DBI::dbWriteTable(Connection, "creatorTypes",
                  data.frame(creatorTypeID = 1L, creatorType = "author"))
```

```
DBI::dbWriteTable(Connection, "creators", data.frame(
  creatorID = 1L, firstName = "Ada", lastName = "Lovelace"
))
DBI::dbWriteTable(Connection, "itemCreators", data.frame(
  itemID = 1L, creatorID = 1L, creatorTypeID = 1L, orderIndex = 0L
))
DBI::dbWriteTable(Connection, "itemAttachments", data.frame(
  itemID = integer(), parentItemID = integer(), linkMode = integer(),
  contentType = character(), path = character(), storageHash = character()
))
DBI::dbWriteTable(Connection, "deletedItems", data.frame(itemID = integer()))
DBI::dbDisconnect(Connection)
zotLocalRecords(path = Path)
unlink(Path)
```

`zotPreview` *Preview a batch plan*

Description

Validates and summarizes a plan for execution by `zotExecute()`. A row has action `"patch"`, `"create"`, `"attach"`, or `"trash"`; a user or group library; an item key for patch and Trash rows; and a payload list.

Usage

```
zotPreview(plan, describe)
```

Arguments

`plan`	A data.table with columns `action`, `library`, `itemKey`, `payload` (list column).
`describe`	Text identifying the batch.

Details

`zotPreview()` validates the plan, rejects unknown actions and recognized source labels, copies the rows, and calculates a digest from a canonical JSON representation. It does not contact Zotero.

Three rules protect the ledger and the write itself. A create row leaves `itemKey` as `NA`, because `zotExecute()` identifies a create by its position and a keyed create would make a resumed batch create the item again. An identified item appears once per library, because the ledger holds one entry per library and key and a repeated pair would drop the second row. A payload carries no `key` or `version` field: `zotPatch()` sends the version as a conditional header, and a payload version is read back changed, which reports a successful write as failed.

An attach row creates an attachment item and uploads a local file; it leaves `itemKey` as `NA`, like a create row. Its payload carries `parentItem` (`NULL` for a standalone attachment), the local `path`, the stored `filename`, `contentType`, an optional `charset`, the file's `md5` (32 lowercase hexadecimal digits) and `filesize` (a whole number of bytes above zero); `ztAttachPlan()` builds such rows. Source labels are refused in `filename` only, because the local path is never stored in Zotero. The digest covers the whole payload, so an approval binds the exact bytes through their MD5.

Value

A zotPreview: the plan, its canonical SHA-256 digest, counts by action and library (N rows and the bytes attach rows upload, 0 for other actions) and creation time.

See Also

zotApprove(), zotExecute(), zotVerify(), and ztAttachPlan().

Examples

```
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "ABCD1234",
  payload = list(list(title = "A reviewed title"))
)
Preview <- zotPreview(Plan, describe = "Correct one title")
Preview
```

---

zotReadBatch *Read a batch file written by zot*

Description

Reads a plan, preview, approval or ledger file and rebuilds the corresponding object. A preview is rebuilt through zotPreview(), so its digest is recalculated from the rows on disk and compared with the stored one: a file edited after it was written is refused.

Usage

```
zotReadBatch(path, digest = NULL)
```

Arguments

path	A file written by zotWriteBatch() or zotExecute().
digest	Optional expected SHA-256 digest; a mismatch is refused.

Details

A ledger has no kind; it is recognized by its digest and items and returned as the zotLedger list that zotExecute() produces. The ledger's optional schema is preserved; reading a legacy ledger does not migrate it or authorize its execution. A plan has no stored digest; when digest is supplied for a plan it is compared with the digest the plan would have under zotPreview().

Value

A plan data.table, a zotPreview, a zotApproval or a zotLedger.

See Also

zotWriteBatch(), zotExecute().

Examples

```
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "ABCD1234",
  payload = list(list(title = "A reviewed title"))
)
Preview <- zotPreview(Plan, describe = "Correct one title")
Path <- tempfile(fileext = ".json")
zotWriteBatch(Preview, Path)
identical(zotReadBatch(Path)$digest, Preview$digest)
unlink(Path)
```

---

zotRequest *Perform one Zotero API request*

Description

Core transport for every zot API call: bounded retries, Backoff and Retry-After honoring, and conditional writes through If-Unmodified-Since-Version. Returns for every HTTP status — callers interpret status (a 404 is a verdict, not a transport failure); only an exhausted transport raises a zotHttpError.

Usage

```
zotRequest(config, method, path, query = list(), body = NULL, version = NULL)

zotGet(config, path, query = list())

zotPost(config, path, body)

zotPatch(config, path, body, version)
```

Arguments

config	A zotConfig() object.
method	HTTP method: "GET", "POST" or "PATCH".
path	API path starting at the host root (see zotLibraryPath()).
query	Named list of query parameters.
body	Request body, serialized as JSON when not NULL.
version	Value for If-Unmodified-Since-Version; required by zotPatch() so no write is unconditional.

Details

A performer receives `config`, `method`, `path`, `query`, `body`, and `version`, and returns a list with `status`, `headers`, and `body`. This makes the complete HTTP boundary replaceable in examples and tests without changing the public request contract. Responses eligible for retry are bounded by five attempts; other HTTP statuses are returned unchanged. POST is retried only after rate limiting (HTTP 429). A server failure on POST can follow an effect, so it is returned without another write; the guarded executor retains that unknown outcome as an attempt. GET and versioned PATCH retain their retries on HTTP 500, 502 and 503.

Value

A list with `status`, `headers` (named list) and `body` (parsed JSON, or `NULL` when empty).

See Also

`zotGetAll()`, `zotTrash()`, and `zotPreview()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  list(
    status = if (method == "POST") 201L else 200L,
    headers = list(),
    body = list(method = method, path = path, body = body, version = version)
  )
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
zotGet(Config, path = "/users/42/items/ABCD1234")$status
zotPost(Config, path = "/users/42/items", body = list(list(title = "New")))$status
zotPatch(Config, path = "/users/42/items/ABCD1234",
         body = list(title = "Revised"), version = 7L)$body$version
```

`zotSameItem` *Strong-identity comparison of two bibliographic records*

Description

A duplicate is only what strong identity proves: an exact DOI, an exact attachment MD5, or normalized title plus year plus first-author family name. A bare title match proves nothing — generic titles collide across distinct documents — and reuse is refused when the first record carries a file the second lacks, whatever the metadata says.

Usage

```
zotSameItem(a, b)
```

Arguments

a, b	Lists with any of `doi`, `md5`, `title`, `year`, `author` (first-author family name) and `hasFile` (logical).

Value

A list: `match` (logical) and `basis` (`"doi"`, `"md5"`, `"title-year-author"` or `"none"`).

See Also

`ztTwins()` for pairwise candidate detection.

Examples

```
A <- list(doi = "10.1000/Example", hasFile = TRUE)
B <- list(doi = "10.1000/example", hasFile = TRUE)
zotSameItem(A, B)

# A title alone never establishes identity.
zotSameItem(list(title = "Introduction"), list(title = "Introduction"))
```

---

zotStatus *Probe the data planes zot can reach*

Description

Reports, one row per plane, whether the installed package, the API credentials, the Web API and the local database are available from this process, with the observation behind each verdict. The API probe is one `limit = 1` read of the personal library; the local probe opens the database read-only and counts its items.

Usage

```
zotStatus(
  config = NULL,
  path = file.path(Sys.getenv("HOME"), "Zotero", "zotero.sqlite")
)
```

Arguments

config	A `zotConfig()` object, or `NULL` to read the credentials from the environment.
path	Local database file, as in `zotLocalQuery()`.

Details

The credentials row never contains a credential: when a variable is missing, `detail` names it and the file where it is defined, as `zotConfig()` does. The local plane is advisory; the Web API is the plane that writes are verified against.

Value

A data.table with `plane`, `available` and `detail`.

See Also

`zotConfig()`, `zotLocalQuery()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  list(status = 200L, headers = list(`last-modified-version` = "9"),
       body = list())
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
zotStatus(Config, path = tempfile())
```

---

zotTrash *Move items to the recoverable Trash*

Description

The only removal zot implements: `deleted = 1` through a minimal conditional PATCH. Permanent deletion is not part of this package; emptying Trash is not provided by this package.

Usage

```
zotTrash(config, keys, library = "user")
```

Arguments

`config`	A `zotConfig()` object.
`keys`	Character vector of item keys.
`library`	`"user"` or a group id (see `zotLibraryPath()`).

Value

A data.table with one row per key: `itemKey`, `status` (`"trashed"`, `"already"`, `"missing"` or `"failed"`).

See Also

`zotPatch()`, `zotPreview()`, and `zotExecute()`.

Examples

```
Store <- new.env(parent = emptyenv())
Store$version <- 1L
Store$deleted <- 0L
Performer <- function(config, method, path, query, body, version) {
  if (method == "GET") {
    return(list(
      status = 200L,
      headers = list(),
      body = list(version = Store$version,
                  data = list(deleted = Store$deleted))
    ))
  }
  Store$deleted <- body$deleted
  Store$version <- Store$version + 1L
  list(status = 204L, headers = list(), body = NULL)
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
zotTrash(Config, keys = "ABCD1234")
```

`zotVerify` *Census a ledger against its plan*

Description

Summarizes item statuses and re-reads completed rows: a bounded sample of patches by default, or every completed row when `full` is `TRUE`.

Usage

```
zotVerify(config, preview, ledger, full = FALSE)
```

Arguments

`config`	A `zotConfig()` object.
`preview`	The executed `zotPreview()`.
`ledger`	The `zotExecute()` ledger.
`full`	Re-read every completed row instead of sampling patches.

Details

By default at most ten completed patch rows are sampled and create, attach and Trash rows are not included. With `full = TRUE` every completed row is re-read: a patch and a create compare the payload fields against the current record, an attach row compares the MD5, filename, content type and parent of the attachment, and a Trash row confirms `deleted`. A created item is resolved through the key its ledger entry recorded. Rows whose current state does not confirm the plan are listed in `unconfirmed`; the result does not replace inspection of failed ledger entries. Missing plan rows are counted as `"pending"`. Schema and legacy identity checks are the same as `zotExecute()`.

Value

A list: `counts` by status, `sampleChecked`, `sampleConfirmed` and `unconfirmed` (the ledger ids that did not confirm), `unfinished` (plan ids without a done entry), and `complete` (all plan rows done and none of the checked rows unconfirmed). Completeness does not turn a sample into a full read-back.

See Also

`zotExecute()` and `ztCensus()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  list(
    status = 200L, headers = list(),
    body = list(version = 2L, data = list(title = "After"))
  )
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "K1",
  payload = list(list(title = "After"))
)
Preview <- zotPreview(Plan, describe = "Synthetic verified patch")
Ledger <- structure(
  list(schema = 2L, digest = Preview$digest,
       items = list(`user:K1` = list(status = "done"))),
  class = "zotLedger"
)
zotVerify(Config, Preview, Ledger)
```

`zotWriteBatch` *Write a plan, preview or approval as a batch file*

Description

Persists a guarded-flow object as JSON so that planning, approval and execution can happen in separate processes. The file carries a `kind` (`"zotPlan"`, `"zotPreview"` or `"zotApproval"`) and the fields of the corresponding object; a preview stores its rows together with its digest.

Usage

```
zotWriteBatch(x, path)
```

Arguments

`x`	A plan (a data.table with `action`, `library`, `itemKey` and `payload`), a `zotPreview()` or a `zotApprove()` record.
`path`	Destination file.

Details

The file is written to a temporary sibling and renamed into place, so a reader never observes a partially written batch file. An existing file at `path` is replaced. Ledgers are written by `zotExecute()`, not here.

Value

`path`, invisibly.

See Also

`zotReadBatch()`, `zotPreview()`, `zotApprove()`.

Examples

```
Plan <- data.table::data.table(
  action = "patch", library = "user", itemKey = "ABCD1234",
  payload = list(list(title = "A reviewed title"))
)
Path <- tempfile(fileext = ".json")
zotWriteBatch(Plan, Path)
zotReadBatch(Path)
unlink(Path)
```

ztAttach *Attach a local file to an existing item*

Description

Implements Zotero's full upload protocol: create the attachment item, request upload authorization (an identical file already in storage short-circuits as `exists`), upload to the returned storage URL, register the upload, and verify the attachment's MD5 server-side. Uploads may consume Zotero storage quota and have immediate external effects.

Usage

```
ztAttach(
  config,
  parentKey,
  filePath,
  library = "user",
  filename = basename(filePath),
  contentType = .zotContentType(filePath),
  fileTransport = config$fileTransport
)
```

Arguments

`config`	A `zotConfig()` object.
`parentKey`	Key of the regular item receiving the file, or `NA` for a standalone attachment.
`filePath`	Local file to upload.
`library`	`"user"` or a group id.
`filename`	Stored filename; defaults to the file's basename.
`contentType`	MIME type; defaults to Zotero's choice by extension table and text/binary sniff (see Details).
`fileTransport`	The three protocol steps as an injectable list (`authorize`, `upload`, `register`); defaults to the `fileTransport` of `config`, and to the real protocol when the configuration has none. Tests replace it.

Details

The default file transport performs every external protocol step. Supplying `fileTransport` replaces only authorization, byte upload, and registration; attachment creation and final API verification still use `config`. A stored file with the same MD5 returns `"exists"` before uploading bytes.

The default content type follows Zotero's desktop client: a table of known extensions (`pdf`, `epub`, `md`, `bib`, `txt`, `html`, `csv`, `json`, office formats, images and others), and for any other extension a sniff of the first 1024 bytes, which yields `text/plain` when every byte is a text character and `application/octet-stream` otherwise. An empty file has no type and is refused with a `zotPlanError`. A `text/*` file whose bytes are valid UTF-8 is created with `charset = "utf-8"`. The guarded lane offers the same upload with preview, approval and a resumable ledger through `ztAttachPlan()` and the `attach` action of `zotExecute()`.

Value

A list: `status` (`"uploaded"`, `"exists"` or `"failed"`), the attachment `key`, and `error` when failed.

See Also

`ztCleanName()`, `ztAttachPlan()`, `zotPreview()`, and `zotExecute()`.

Examples

```
Path <- tempfile(fileext = ".pdf")
writeBin(as.raw(1:8), Path)
Performer <- function(config, method, path, query, body, version) {
  list(
    status = 200L, headers = list(),
    body = list(successful = list(`0` = list(key = "ATT1")))
  )
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
Transport <- list(
  authorize = function(...) list(exists = 1L),
  upload = function(...) stop("upload should not run"),
  register = function(...) stop("registration should not run")
)
ztAttach(Config, parentKey = NA_character_, filePath = Path,
         fileTransport = Transport)
unlink(Path)
```

`ztAttachPlan` *Build an attach plan for local files*

Description

Produces one `attach` row per local file for the guarded flow: `zotPreview()`, `zotApprove()`, `zotExecute()` and `zotVerify()`. Files whose MD5 is already an attachment of the same parent are omitted.

Usage

```
ztAttachPlan(config, files, library = "user")
```

Arguments

config	A zotConfig() object.
files	A data.table with path and parentKey (NA for a standalone attachment) and, optionally, filename, contentType and charset; NA in an optional column takes the default.
library	"user" or a group id.

Details

Every file is checked before any request: a missing or empty file is refused with a zotPlanError. Each row records the file's md5 and filesize, so the approval of its preview binds the exact bytes. The stored filename is cleaned with ztCleanName(); contentType and charset default to the choice ztAttach() documents. The path is kept as given, and a relative path is resolved against the working directory of execution.

Planning reads the children of each distinct parent once through zotListItems() and omits the rows whose MD5 is already attached there, with a message per omitted file. A parent whose children exceed one listing page is refused with a zotBoundError, because the omission could not see every attachment. Standalone files are not compared. Planning does not write.

Value

A guarded-flow plan of attach rows, empty when every file is already attached.

See Also

ztAttach(), zotPreview(), and zotExecute().

Examples

```
Path <- tempfile(fileext = ".md")
writeLines("# Notes", Path)
Performer <- function(config, method, path, query, body, version) {
  list(status = 200L,
       headers = list(`total-results` = "0", `last-modified-version` = "7"),
       body = list())
}
Config <- zotConfig(userID = "42", key = "example", performer = Performer)
Files <- data.table::data.table(path = Path, parentKey = "PARENT1")
Plan <- ztAttachPlan(Config, files = Files, library = "user")
Plan$payload[[1L]]$contentType
unlink(Path)
```

ztBlockMoveCreate *Build the create half of a block move*

Description

Produces create rows in the destination library from the given records. Items carrying imported files are refused: moving files needs the upload protocol, which this plan builder does not perform.

Usage

```
ztBlockMoveCreate(records, toLibrary, collectionKey)
```

Arguments

records	A data.table with itemKey, payload (list column of item fields) and hasFile (logical).
toLibrary	Destination library.
collectionKey	Destination collection key.

Value

A guarded-flow plan of create rows, one per record, whose order matches records.

See Also

ztBlockMoveTrash(), zotPreview(), and ztAttach().

Examples

```
Records <- data.table::data.table(
  itemKey = "OLD1",
  payload = list(list(itemType = "book", title = "Reviewed book")),
  hasFile = FALSE
)
ztBlockMoveCreate(Records, toLibrary = "5107760", collectionKey = "DEST")
```

ztBlockMoveTrash *Build the trash half of a block move from its executed create ledger*

Description

Returns Trash rows for originals whose corresponding create-ledger entry is marked "done" and records a new item key.

Usage

```
ztBlockMoveTrash(records, createLedger, fromLibrary)
```

Arguments

records	The records given to ztBlockMoveCreate().
createLedger	The zotExecute() ledger of the create plan.
fromLibrary	Library holding the originals.

Value

A guarded-flow plan of trash rows.

See Also

ztBlockMoveCreate(), zotExecute(), and zotTrash().

Examples

```
Records <- data.table::data.table(
  itemKey = "OLD1", payload = list(list(title = "Reviewed book")),
  hasFile = FALSE
)
Ledger <- list(items = list(`create-1` = list(status = "done", new = "NEW1")))
ztBlockMoveTrash(Records, createLedger = Ledger, fromLibrary = "user")
```

---

ztCensus *Census a library through the API*

Description

Returns top-level and Trash counts for one library, plus per-collection counts when collection keys are supplied.

Usage

```
ztCensus(config, library = "user", collectionKeys = NULL)
```

Arguments

config	A zotConfig() object.
library	"user" or a group id.
collectionKeys	Optional named character vector: name -> key.

Details

Each scope is counted from the API Total-Results response header while requesting only one record. Supplying named collection keys adds one row per requested collection without downloading the library contents.

Value

A data.table with scope and total.

See Also

zotGet(), zotLocalItems(), and zotVerify().

Examples

```
Performer <- function(config, method, path, query, body, version) {
  Total <- if (grepl("/trash", path)) "7" else "100"
  list(status = 200L, headers = list(`total-results` = Total), body = list())
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
ztCensus(Config)
```

---

ztCheckPayload *Check a create payload against its item type template*

Description

Verifies what the campaigns checked by hand before creating records: every field exists for the type, text fields are single strings, array and object fields have the template's shape, each creator has an accepted type and either a single name or both firstName and lastName, and no text still carries an acquisition-site label. Problems are reported together in one zotPlanError.

Usage

```
ztCheckPayload(
  config,
  payload,
  template = ztTemplate(config, payload$itemType)
)
```

Arguments

config	A zotConfig() object.
payload	A create payload (named list) with an itemType.
template	The item type's ztTemplate(); fetched when absent.

Value

Invisibly, a list with itemType, fields and creators (their count) when the payload passes.

See Also

ztTemplate(), ztCleanPayload(), zotPreview().

Examples

```
Template <- list(
  itemType = "book", fields = c("itemType", "title", "creators", "tags"),
  creatorTypes = "author",
  template = list(itemType = "book", title = "", creators = list(), tags = list())
)
Payload <- list(itemType = "book", title = "A book",
                creators = list(list(creatorType = "author", name = "Team")))
ztCheckPayload(config = NULL, payload = Payload, template = Template)
```

---

ztCleanName *Normalize an incoming filename*

Description

Removes source labels recognized by the package and copy markers such as (1) before the extension. The extension and other parenthetical text are retained.

Usage

```
ztCleanName(name)
```

Arguments

name	A filename.

Value

The cleaned filename.

See Also

ztCleanTitle() and ztCleanPayload().

Examples

```
ztCleanName("Dynamics (1).pdf")
ztCleanName("Course Notes (SEG Course).pdf")
```

ztCleanPayload *Sanitize every text field of an incoming item payload*

Description

Applies the package's text-normalization rules across an item payload. Titles lose trailing extensions and copy markers, filenames retain their extensions, and creator names and tags are included. Non-text values are returned unchanged.

Usage

```
ztCleanPayload(payload)
```

Arguments

payload	An item payload (named list of fields).

Value

The payload with every text field cleaned.

See Also

ztCleanName(), ztCleanTitle(), and zotPreview().

Examples

```
Payload <- list(
  itemType = "book",
  title = "Structural Analysis (1).pdf",
  creators = list(list(creatorType = "author", firstName = "A.",
                       lastName = "Author")),
  filename = "Structural Analysis (1).pdf"
)
ztCleanPayload(Payload)
```

ztCleanTitle *Normalize an incoming title*

Description

Like ztCleanName(), and additionally strips a trailing file extension and trailing copy markers — a title is not a filename.

Usage

```
ztCleanTitle(title)
```

Arguments

`title`	An item title.

Value

The cleaned title.

See Also

`ztCleanName()`, `ztCleanPayload()`, and `ztRetitle()`.

Examples

```
ztCleanTitle("Dynamics of Structures (1).pdf")
ztCleanTitle("Optimization (2nd Edition)")
```

---

ztCollectionTree *Ensure a collection tree exists in a library*

Description

Find or create each path (vectors of names, root first) and return the full path to key map. Creating a missing collection has an immediate API effect.

Usage

```
ztCollectionTree(config, library, paths)
```

Arguments

`config`	A `zotConfig()` object.
`library`	`"user"` or a group id.
`paths`	A list of character vectors, e.g. `list(c("Conferences", "ICOLD11"))`.

Details

Existing collections are indexed by parent key and name. Missing nodes are created from the root downward and reused by later paths in the same call. Unlike the plan builders, this function has immediate external effects.

Value

Named character vector: `"A/B"` -> collection key.

See Also

`ztRefile()` and `zotPreview()`.

Examples

```
Counter <- new.env(parent = emptyenv())
Counter$n <- 0L
Performer <- function(config, method, path, query, body, version) {
  if (method == "GET") {
    return(list(
      status = 200L, headers = list(`total-results` = "0"), body = list()
    ))
  }
  Counter$n <- Counter$n + 1L
  Key <- paste0("COLL", Counter$n)
  list(
    status = 200L, headers = list(),
    body = list(successful = list(`0` = list(key = Key)))
  )
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
ztCollectionTree(Config, library = "user", paths = list(c("Projects", "2026")))
```

---

ztCrossref *Map a Crossref record to a Zotero item payload*

Description

The returned payload is cleaned with `ztCleanPayload()` and can be placed in a create plan. The default fetcher contacts the public Crossref API; inject a compatible function for reproducible or offline work.

Usage

```
ztCrossref(doi, getJson = .zotCrossrefGet)
```

Arguments

doi	The DOI to resolve.
getJson	Injectable fetcher returning the parsed Crossref message for a DOI; the default performs the real request.

Value

An item payload for a create row, or a classed error when the DOI does not resolve.

See Also

`ztCleanPayload()` and `zotPreview()`.

Examples

```
Message <- function(doi) list(
  title = list("A reproducible record"),
  author = list(list(given = "Ada", family = "Lovelace")),
  `container-title` = list("Example Journal"),
  issued = list(`date-parts` = list(list(2026L)))
)
ztCrossref("10.1000/example", getJson = Message)
```

`ztMatches` *Match candidate records against a set of records by strong identity*

Description

Nominates pairs whose normalized DOI, MD5 or title coincide and judges each pair with `zotSameItem()`. The result reports every nominated pair with its verdict, as `ztTwins()` does within one set, so a nominated title collision that is not a match remains visible.

Usage

```
ztMatches(candidates, records)
```

Arguments

`candidates`, `records`
: Lists of records, or data.frames with one record per row, in any shape `zotIdentity()` accepts.

Value

A data.table with `index` (position in `candidates`), `candidate` (its key, or `NA`), `itemKey`, `library`, `match` and `basis`.

See Also

`zotIdentity()`, `zotSameItem()`, `ztTwins()`.

Examples

```
Candidates <- list(list(title = "Paper", year = "2026", author = "Lovelace"))
Records <- list(
  list(key = "R1", library = list(type = "user"),
       data = list(title = "Paper", date = "2026",
                   creators = list(list(creatorType = "author",
                                        lastName = "Lovelace")))),
  list(key = "R2", data = list(title = "Paper", date = "1999"))
)
ztMatches(Candidates, Records)
```

`ztRefile` *Build a refile plan that adds items to a collection*

Description

Fetches each item's current memberships at plan time and produces patch rows whose `collections` payload is the union with `addTo` — the Zotero API replaces the whole membership list, so the merge is mandatory.

Usage

```
ztRefile(config, itemKeys, library, addTo)
```

Arguments

`config`	A `zotConfig()` object.
`itemKeys`	Character vector of item keys.
`library`	`"user"` or a group id.
`addTo`	Collection key to add.

Details

Planning performs one current API read per key. It does not write, and it omits missing items and items already in the destination collection. Memberships can change after planning; rebuild or review the plan before execution when concurrent changes are possible.

Value

A guarded-flow plan (items already members are omitted).

See Also

`ztCollectionTree()`, `zotPreview()`, and `zotExecute()`.

Examples

```
Memberships <- list(K1 = list("OLD"), K2 = list("TARGET"))
Performer <- function(config, method, path, query, body, version) {
  Key <- sub(".*/items/", "", path)
  list(
    status = 200L, headers = list(),
    body = list(version = 1L, data = list(collections = Memberships[[Key]]))
  )
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
```

```
)
ztRefile(Config, itemKeys = c("K1", "K2"), library = "user",
         addTo = "TARGET")
```

`ztRetitle` *Build a retitle plan from a deterministic transformation*

Description

Pure plan builder: applies `transform` to every title and returns patch rows only where the result differs. Feed the result to `zotPreview()`.

Usage

```
ztRetitle(items, transform)
```

Arguments

`items`	A data.table with `itemKey`, `library`, `title`.
`transform`	A function from one title to its cleaned form.

Value

A guarded-flow plan (possibly empty).

See Also

`ztCleanTitle()` and `zotPreview()`.

Examples

```
Items <- data.table::data.table(
  itemKey = c("A", "B"), library = "user",
  title = c("Reviewed book (1).pdf", "Existing title")
)
ztRetitle(Items, transform = ztCleanTitle)
```

`ztTemplate` *Fetch the field template of an item type*

Description

Reads the empty item the API returns for a type and the creator types it accepts. The field names are the complete set a create payload may carry; the template values show each field's shape (text, array or object).

Usage

```
ztTemplate(config, itemType)
```

Arguments

`config`	A `zotConfig()` object.
`itemType`	An API item type such as `"book"` or `"journalArticle"`.

Value

A list: `itemType`, `fields`, `creatorTypes` and `template`.

See Also

`ztCheckPayload()`, `ztCrossref()`, `zotPreview()`.

Examples

```
Performer <- function(config, method, path, query, body, version) {
  if (path == "/items/new") {
    return(list(status = 200L, headers = list(),
                body = list(itemType = "book", title = "", creators = list(),
                            tags = list(), collections = list())))
  }
  list(status = 200L, headers = list(),
       body = list(list(creatorType = "author", localized = "Author")))
}
Config <- structure(
  list(userID = "42", key = "example", apiBase = "https://example.invalid",
       performer = Performer),
  class = "zotConfig"
)
ztTemplate(Config, itemType = "book")$fields
```

`ztTwins` *Detect duplicate candidates with evidence classes*

Description

Groups records by normalized title to nominate candidates, then judges every pair with `zotSameItem()` — the strong-identity rule decides, never the title alone.

Usage

```
ztTwins(records)
```

Arguments

`records`	A data.table with `itemKey` and any of `title`, `year`, `author`, `doi`, `md5`, `hasFile`.

Value

A data.table of pairs: `keyA`, `keyB`, `match`, `basis`.

See Also

`zotSameItem()` and `zotPreview()`.

Examples

```
Records <- data.table::data.table(
  itemKey = c("A", "B"), title = c("Paper", "Paper"),
  year = c(2026L, 2026L), author = c("Lovelace", "Lovelace"),
  doi = NA_character_, md5 = NA_character_, hasFile = FALSE
)
ztTwins(Records)
```

Index

zotApprove, 2
zotApprove(), *5, 6*, *19*, *26*, *28*
zotCollections, 3
zotCollections(), *10*
zotConfig, 4
zotConfig(), *3*, *5*, *7*, *9–12*, *20*, *22–24*, *27*, *29*, *31, 32*, *35*, *38*, *40*
zotExecute, 5
zotExecute(), *2, 3*, *5*, *18–20*, *23–29*, *31*, *38*
zotGet *(*zotRequest*)*, 20
zotGet(), *7*, *9*, *32*
zotGetAll, 7
zotGetAll(), *12*, *21*
zotIdentity, 8
zotIdentity(), *17*, *37*
zotItems, 9
zotItems(), *12*
zotLibraries, 10
zotLibraryPath, 11
zotLibraryPath(), *7*, *10*, *20*, *23*
zotListItems, 11
zotListItems(), *9*, *29*
zotLocalFiles, 13
zotLocalItems, 14
zotLocalItems(), *16, 17*, *32*
zotLocalQuery, 15
zotLocalQuery(), *14–16*, *22, 23*
zotLocalRecords, 16
zotLocalRecords(), *8*, *14*
zotPatch *(*zotRequest*)*, 20
zotPatch(), *18*, *20*, *23*
zotPost *(*zotRequest*)*, 20
zotPreview, 18
zotPreview(), *3*, *5, 6*, *19*, *21*, *23, 24*, *26*, *28–30*, *33–36*, *38–41*
zotReadBatch, 19
zotReadBatch(), *26*
zotRequest, 20
zotRequest(), *5*, *7*, *11*, *16*
zotSameItem, 21
zotSameItem(), *8*, *14*, *16*, *37*, *41*
zotStatus, 22
zotTrash, 23
zotTrash(), *6*, *21*, *31*
zotVerify, 24
zotVerify(), *6*, *9*, *19*, *28*, *32*
zotWriteBatch, 26
zotWriteBatch(), *19, 20*
ztAttach, 27
ztAttach(), *5*, *14*, *29, 30*
ztAttachPlan, 28
ztAttachPlan(), *18, 19*, *27, 28*
ztBlockMoveCreate, 30
ztBlockMoveCreate(), *31*
ztBlockMoveTrash, 30
ztBlockMoveTrash(), *30*
ztCensus, 31
ztCensus(), *4*, *10*, *12*, *15*, *25*
ztCheckPayload, 32
ztCheckPayload(), *40*
ztCleanName, 33
ztCleanName(), *28, 29*, *34, 35*
ztCleanPayload, 34
ztCleanPayload(), *33*, *35, 36*
ztCleanTitle, 34
ztCleanTitle(), *33, 34*, *39*
ztCollectionTree, 35
ztCollectionTree(), *3, 4*, *38*
ztCrossref, 36
ztCrossref(), *40*
ztMatches, 37
ztMatches(), *8*, *17*
ztRefile, 38
ztRefile(), *4*, *35*
ztRetitle, 39
ztRetitle(), *35*
ztTemplate, 40
ztTemplate(), *32, 33*

ztTwins, 41
ztTwins(), *22*, *37*