

Package: jdv (via r-universe)

October 21, 2026

Title Lightweight Data Viewer

Version 0.0.9000

Description A lightweight data viewer with no package dependencies. It sends data frames, lists, and other R objects, as well as 'Apache Arrow' data, over a local connection to an external viewer application, and answers the viewer's requests for sorted, filtered, and paged rows. Arrow data is read through the Arrow C stream interface and sent as Arrow IPC without conversion to R vectors.

License MIT + file LICENSE

URL <https://github.com/eitsupi/jdv>

BugReports <https://github.com/eitsupi/jdv/issues>

Suggests arrow, bit64, callr, hms, jsonlite, methods, nanoarrow, polars, testthat (>= 3.0.0), tibble, vctrs, withr

Additional_repositories <https://community.r-multiverse.org>

Config/testthat/edition 3

Copyright file inst/COPYRIGHTS

Encoding UTF-8

NeedsCompilation yes

Roxygen list(markdown = TRUE)

RoxygenNote 8.0.0

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

Date/Publication 2026-10-21 14:32:24 UTC

RemoteUrl <https://github.com/eitsupi/jdv>

RemoteRef HEAD

RemoteSha a7506ef2215e48dd3646ca347eaa3c166416107e

RemoteSubdir r-pkg

Contents

jdv_binding	2
jdv_close	3
jdv_connect	4
jdv_errors	5
jdv_handle	6
jdv_notifications	9
jdv_query_spec	10
jdv_register	19
jdv_rpc	21
jdv_spec	22
jdv_view	45
Index	48

jdv_binding	<i>Find the binding behind a variable name</i>
-------------	--

Description

`jdv_binding()` looks up a variable name without evaluating anything and returns the value for the binding argument of `jdv_register()`. It is meant for code that shows an object on behalf of a caller, such as a replacement for `View()`, so that the view follows the caller's variable as `jdv_view()` does. See the Details of `jdv_view()` for what tracking detects.

Usage

```
jdv_binding(name, env = parent.frame())
```

Arguments

name	The name to look up: one string or a symbol, for example <code>substitute(x)</code> . An expression that is not a name, such as a call or a constant, cannot be tracked and gives <code>NULL</code> .
env	The environment to start the search from. The search continues outward through its parents, and the first environment that has a binding for <code>name</code> is used, as <code>jdv_view()</code> does.

Details

Nothing is evaluated: a promise is not forced and an active binding is not called. To track an argument of the calling function, force it first, for example by evaluating it before the call, or `jdv_binding()` returns `NULL` for it.

A non-`NULL` result means the binding can be read without evaluation. `jdv_register()` additionally requires that the binding holds exactly the object registered and that the object is not a sequential source; it fails otherwise, so pass the binding only together with the object the variable holds.

A name that is not a single non-empty string or a symbol, or an `env` that is not an environment, is an error with class `jdv_error_invalid_argument`; see `jdv_errors`.

Value

A list `list(name = , env = )` that can be passed to the `binding` argument of `jdv_register()` as it is, where `env` is the environment in which the binding was found. It is `NULL`, without a warning, when the name cannot be tracked: the expression is not a name, no environment from `env` outward has the binding, the binding is a promise that has not been forced, an active binding, or an empty argument, or R is older than 4.6.0.

Examples

```
df <- data.frame(a = 1:3)
binding <- jdv_binding("df")
identical(binding$env, globalenv())

# A viewer hook that follows the variable its caller passed.
view <- function(x) {
  jdv_register(
    x,
    title = deparse1(substitute(x)),
    binding = jdv_binding(substitute(x), parent.frame())
  )
}
view(df)
jdv_disconnect()
```

jdv_close

*Close one or all views in the local jdv viewer***Description**

Close one or all views in the local jdv viewer

Usage

```
jdv_close(id = NULL)
```

Arguments

<code>id</code>	Optional view id returned by <code>jdv_view()</code> . When omitted, all active views are closed.
-----------------	---

Details

Closing the last table of a connection opened implicitly by `jdv_view()` stops the transport event source and closes the local connection, after the `view/closed` notification (and, for a viewer request, its response) has been sent. A persistent connection from `jdv_connect()` stays open. A close requested from inside a viewer callback is safe; native source cleanup is deferred until the callback returns. In an embedded session (see `jdv_notifications()`) the `view/closed` messages are queued for the host and the session stays open.

Value

The ids that were closed, invisibly.

jdv_connect	<i>Manage the local jdv viewer connection</i>
-------------	---

Description

[jdv_connect\(\)](#) opens the local viewer connection and sends hello immediately, even when no view is open. Such an explicit connection is persistent: it stays open, and keeps handling viewer requests, after the last view is closed. [jdv_disconnect\(\)](#) closes the connection and every view. [jdv_connection\(\)](#) reports the current state.

Usage

```
jdv_connect(address = NULL)
```

```
jdv_disconnect()
```

```
jdv_connection()
```

Arguments

address	Optional viewer address. When omitted, the address is selected from <code>options(jdv.address = ...)</code> , then <code>JDV_ADDRESS</code> , then <code>discovery.json</code> under the user cache directory (see jdv_view()).
---------	--

Details

A connection opened implicitly by [jdv_view\(\)](#) is closed after its last view is closed. Calling [jdv_connect\(\)](#) while such a connection is open promotes it to a persistent connection without sending hello again. Calling [jdv_connect\(\)](#) again for the same address, or without an address while connected, changes nothing. Asking for a different address while connected is an error: call [jdv_disconnect\(\)](#) first. A later [jdv_view\(\)](#) uses the existing connection and does not repeat hello.

A peer disconnect, a failed send, or package unload ends the session. jdv does not reconnect automatically, and views are not resent to a later session. [jdv_disconnect\(\)](#) is idempotent and also ends an embedded session, releasing views registered with [jdv_register\(\)](#) and discarding its queued notifications. Ownership never moves implicitly: while an embedded session is active, [jdv_connect\(\)](#) and [jdv_view\(\)](#) fail until [jdv_disconnect\(\)](#) is called. [jdv_connection\(\)](#) has no side effects: it neither connects nor polls.

Errors raised on purpose, such as an unreachable address or an address that conflicts with the current connection, carry condition classes that name the reason; see [jdv_errors](#).

Value

A list describing the connection after the call. `jdv_connect()` and `jdv_disconnect()` return it invisibly. The members are:

- `state`: "disconnected", "embedded" (an embedded notification session is active; see `jdv_notifications()`), "connected", or "closing" (the connection is closed and its cleanup is still pending).
- `connected`: TRUE only in the "connected" state.
- `owner`: who receives notifications: "none", "embedded" (the host calls `jdv_notifications()`), or "transport" (the built-in viewer connection).
- `persistent`: TRUE for an explicit connection made or promoted by `jdv_connect()`.
- `address`: the connected address, or NULL.
- `last_disconnect_reason`: a short string for the most recent disconnect, or NULL if there has been none. Values include "disconnect" (`jdv_disconnect()`), "last_view_closed", "peer_closed", "send_failed", "register_failed", "notification_overflow", and "unloaded".

jdv_errors	<i>Errors signaled by jdv</i>
------------	-------------------------------

Description

Errors that jdv raises on purpose are conditions of class `jdv_error`. Each also has a class that names its reason, `jdv_error_<code>`, and a code field with the same `<code>`, so that a caller can decide what to do without matching the message. The full class vector is `c("jdv_error_<code>", "jdv_error", "error", "condition")`; errors of the query layer, such as those of `jdv_register()`, also have the class `jdv_query_error` in front, and a notification queue overflow has the class `jdv_notification_overflow` (see `jdv_notifications()`). Unexpected R errors, for example from a failing `format()` method of the user, are not wrapped.

Details

The codes below are the ones R functions can signal. `invalid_request`, `unsupported`, `unavailable`, `limit_exceeded`, `schema_changed`, `producer_error`, `view_not_found`, and `in_use` have the same meaning as the protocol error codes of the same name (see `jdv_spec`). `invalid_argument`, `connect_failed`, `not_connected`, and `shutting_down` exist only in the R API; they are never sent to a viewer.

Code

```
invalid_argument
in_use
connect_failed
not_connected
shutting_down
invalid_request
unsupported
```

Signaled when

```
An argument or option has a wrong type or value, such as track or a
Another owner holds the resource. An embedded notification session
The viewer cannot be reached: the discovery file is missing, too large
A message is sent while there is no viewer connection.
A function is called while the package is being unloaded.
The binding of jdv_register() has the right shape but does not h
The operation is not available for the object, such as tracking a sequ
```

unavailable	A binding cannot be read without evaluating it.
limit_exceeded	A configured limit is exceeded, including a notification queue overflow.
schema_changed, producer_error, view_not_found	Errors of the query layer; see jdv_query_spec .

The registry-free handle functions (see [jdv_handle\(\)](#)) use the same conditions. A malformed handle, mode, collation, or max_bytes argument signals `invalid_argument`; malformed request params or typed paths signal `invalid_request`. A closed handle, including a child closed by its parent, signals `view_not_found`. Table fingerprint changes signal `schema_changed`. Sequential sources and unsupported operations on a handle's mode signal `unsupported`. Methods other than `grid/rows` and `object/children` passed to [jdv_handle_request\(\)](#) signal `method_not_found`; use [jdv_handle_open\(\)](#) to open a child. A descriptor or single page item that cannot fit its byte limit signals `limit_exceeded`.

Examples

```
library(jdv)
result <- tryCatch(
  jdv_view(mtcars, track = NA),
  jdv_error_invalid_argument = function(error) error$code
)
result

# Fall back when no viewer is available, and report anything else.
open_view <- function(x) {
  tryCatch(
    jdv_view(x),
    jdv_error_connect_failed = function(error) {
      message("No viewer: ", conditionMessage(error))
      NULL
    },
    jdv_error_in_use = function(error) {
      message("The viewer is busy: ", conditionMessage(error))
      NULL
    }
  )
}
```

jdv_handle

Create an isolated jdv handle

Description

A handle captures a data frame, matrix, or list for local queries. Handle queries do not use the viewer registry, its table limit, or the notification queue, and they work whether or not a viewer connection or embedded session exists. Handles do not track variable bindings or refresh values. Recreate a handle after changing the captured object. A handle is valid only in the current R session; saving it, moving it to another process, or serializing and restoring it is not supported. Treat the returned environment as opaque.

Usage

```

jdv_handle(
  x,
  mode = c("auto", "snapshot", "paged"),
  collation = getOption("jdv.collation", "C")
)

jdv_handle_register(handle, title = NULL, mode = NULL, collation = NULL)

jdv_handle_descriptor(
  handle,
  max_bytes = getOption("jdv.max_message_bytes", 10 * 1024^2)
)

jdv_handle_request(
  handle,
  method,
  params = list(),
  max_bytes = getOption("jdv.max_message_bytes", 10 * 1024^2)
)

jdv_handle_open(
  handle,
  path,
  mode = c("auto", "snapshot", "paged"),
  collation = getOption("jdv.collation", "C")
)

jdv_handle_close(handle)

```

Arguments

x	A data frame, matrix, or list.
mode	For jdv_handle() , the table mode: "auto", "snapshot", or "paged". The default "auto" uses the same selection rules as jdv_register() . For jdv_handle_register() , an explicit mode overrides the handle's original intent; NULL (the default) inherits that intent.
collation	For jdv_handle() , the table collation: "C" or "locale"; by default the value of the <code>jdv.collation</code> option, or "C" when unset. Lists do not use this setting. For jdv_handle_register() , an explicit value overrides the handle's original intent; NULL (the default) inherits that intent.
handle	A handle returned by jdv_handle() or jdv_handle_open() .
title	Optional title for a view registered from a handle. If omitted, or NULL, the handle's title is reused.
max_bytes	Maximum encoded JSON result size in bytes. The effective maximum is also bounded by <code>jdv.max_message_bytes</code> .
method	Either "grid/rows" or "object/children".

params	A named list of parameters for method.
path	A non-empty typed object path as used by object/open.

Details

`jdv_handle_descriptor()` returns the JSON descriptor that corresponds to the parameters of a normal view/open message. A Snapshot descriptor contains the complete captured table and does not support grid/rows. Use `mode = "paged"` when queries should return bounded row pages.

`jdv_handle_request()` accepts "grid/rows" for paged table handles and "object/children" for list handles. Its `params` argument is a named list using the corresponding protocol parameters: `offset`, `limit`, `sort`, `filter`, and zero-based `cols` for rows; or typed `path`, `offset`, `limit`, and `attributes` for children. The handle supplies view and generation itself, so callers must not include those fields. Other methods signal `method_not_found`; use `jdv_handle_open()` to create a child handle.

`max_bytes` limits the encoded JSON result. When a row or child page is too large, jdv reduces its number of rows or children until the result fits. It does not remove selected columns. A descriptor is never truncated; if it does not fit, the call signals `limit_exceeded`. Snapshot descriptors include all table data, so use a paged handle for a bounded preview. The limit must be a positive integer number of bytes and counts the UTF-8 JSON string, not R attributes.

A child opened with `jdv_handle_open()` captures the selected value at that path. Closing a handle closes its children; closing a child leaves its parent and siblings open. Keeping a child alive also keeps its parent alive. Garbage collection closes unreachable handles. Table handles check their captured schema on each operation and signal `schema_changed` if dimensions, column names, or column classes change. They do not detect value-only changes or follow rebinding of the original variable. They retain their source for this check even when the descriptor is a Snapshot. Sequential sources, including Arrow and Polars sources, signal `unsupported`. A closed handle signals `view_not_found`; closing an already closed handle succeeds. A root handle can have at most 32 live descendants at once.

`jdv_handle_register()` registers the captured value as a regular view. It uses the captured value directly, without resolving the handle's path again. The handle's mode and collation intent are used when those arguments are NULL (the default). The returned view has its own lifetime and does not track a variable binding.

The JSON strings returned by descriptor and request operations contain the result object only, without an RPC envelope. Errors are raised as jdv conditions; see [jdv_errors](#). For example, a child handle can be opened at a list element with a typed path:

```
tree <- jdv_handle(list(items = list(answer = 42L)))
child <- jdv_handle_open(tree, list(list(type = "index", index = 1L)))
child_descriptor <- jdv_handle_descriptor(child)
jdv_handle_close(child)
jdv_handle_close(tree)
```

Value

`jdv_handle()` and `jdv_handle_open()` return a `jdv_handle` object. `jdv_handle_register()` returns the same id and generation list as `jdv_register()`. `jdv_handle_descriptor()` and `jdv_handle_request()` return a length-one JSON string. `jdv_handle_close()` returns `invisible(NULL)`.

See Also[jdv_spec](#), [jdv_errors](#)**Examples**

```

handle <- jdv_handle(mtcars, mode = "paged")
descriptor <- jdv_handle_descriptor(handle)
preview <- jdv_handle_request(
  handle,
  "grid/rows",
  list(offset = 0L, limit = 10L, cols = 0:3),
  max_bytes = 64 * 1024
)
jdv_handle_close(handle)

```

jdv_notifications*Take pending notifications for an embedding host*

Description

An embedding host that drives [jdv_rpc\(\)](#) itself owns the notification queue of an embedded session. The session starts with the first [jdv_register\(\)](#) call made while no viewer connection exists, and queues hello before the first view/open. [jdv_notifications\(\)](#) removes and returns everything queued so far, in the order the messages were produced. The session, and its queue, outlive the last view so that the final view/closed messages can be taken; [jdv_disconnect\(\)](#) ends it.

Usage

```
jdv_notifications()
```

Details

Every notification source uses the same queue: registration, closing, eviction when `jdv.max_tables` is reached, closing of child views together with their parent, and the view/open of a child created by object/open. Deliver the messages before the response of a request handled by [jdv_rpc\(\)](#) in the same turn. A queue that outgrows `jdv.notification_max_messages` (default `max(256, 4 * jdv.max_tables)`) or `jdv.notification_max_bytes` (default `4 * jdv.max_message_bytes`), both read when the session starts, ends the session instead of discarding messages: every view is closed, the queue is dropped, `last_disconnect_reason` of [jdv_connection\(\)](#) becomes "notification_overflow", and the operation that overflowed it fails.

Value

A character vector of JSON notification messages in FIFO order, one message per element and without a trailing newline. It is `character()` when nothing is queued, no session has been started, or the session has ended; the call never starts a session. Taking the messages hands delivery to the caller; the call is not an acknowledgement. It is an error when the built-in viewer connection owns the notifications, which are then sent by the transport.

Examples

```
library(jdv)
id <- jdv_register(mtcars)$id
notifications <- jdv_notifications()
length(notifications)
response <- jdv_rpc(sprintf(
  '{"id":1,"method":"view/close","params":{"view":"%s","generation":1}}',
  id
))
jdv_notifications()
jdv_disconnect()
```

jdvd_query_spec	<i>Query protocol and R implementation details</i>
-----------------	--

Description

This specification is organized into four parts: common query protocol, R producer extension, reference host, and browser viewer. The common query contract defines filter and sort semantics, pagination results, and shared conformance data without requiring R. R-specific source access and private query engine details are identified separately. The reference host and browser viewer consume the same wire semantics.

Details

Columns are identified by their **zero-based positions**, so duplicate or unusual column names are valid. On the wire, source_row values are **zero-based positions in the original source order**.

Common protocol

Filter AST:

Filter objects use these exact fields:

Operator	Required fields	Supported columns
and, or	op, args (array of child objects)	Any valid children.
not	op, arg (one child object)	Any valid child.
is_na	op, col	Any column with a valid missingness
contains, starts_with, ends_with, text_equals	op, col, value	string, categorical.
eq, ne, lt, le, gt, ge	op, col, value	integer, number, duration, date
between	op, col, lower, upper	integer, number, duration, date
is_true, is_false	op, col	boolean.
in	op, col, values (array of labels)	categorical.

Example:

```
{
  "op": "and",
  "args": [
    {"op": "contains", "col": 2, "value": "ber"},
    {"op": "between", "col": 0, "lower": 1, "upper": 3},
    {"op": "not", "arg": {"op": "is_na", "col": 4}}
  ]
}
```

Logical operators combine Boolean results elementwise. Empty and is true for every row; empty or is false. not performs ordinary Boolean inversion. Missing source values never match a comparison, membership, Boolean, or text leaf. Consequently, negating such a false leaf includes missing rows: use an explicit not is_na condition when that inclusion is unwanted.

is_na uses the source column's missingness mask, including NaN. It is allowed even when filterable is false, provided extraction yields a valid mask. Missing meaning never comes from display labels such as the literal string "NA".

Being displayable and being filterable are separate. Value-based filters are limited to an explicit list of schema type values, those in the table above; the schema's filterable field is true exactly for them. A column of type list or other has filterable false. It is still displayed, sorted when sortable is true, and answers is_na, but every other operator fails with unsupported ("Column is not filterable."). Mapping source-language values to schema types is producer-specific; see the R column mapping section below for the current R producer.

Text:

Text comparison folds **ASCII A-Z only** to a-z. Non-ASCII characters retain their case: Ä and ä remain distinct. Matching uses fixed literal strings, not regular expressions. Brackets, dots, and asterisks have no pattern meaning. An empty needle matches every nonmissing text for contains/prefix/suffix operations; text_equals with an empty value matches only an empty source string.

Text sources are:

- String: the original, untruncated character value.
- Categorical: its label.

Number, integer, Boolean, duration, date, datetime, and int64 columns do not support text operators, and neither do other columns. List and other composite columns may have previews but no text-filter capability. Page display truncation or formatting does not alter the filter source. Categorical in is exact, case-sensitive label membership and does not use text folding; duplicate labels are harmless and an empty label array matches no rows.

Blank and non-blank filters:

There is no dedicated blank operator; clients combine existing ones. A text-filterable column has a schema type of string or categorical and filterable true. For such a column, blank (missing or empty string) is

```
{ "op": "or", "args": [
  { "op": "is_na", "col": 2 },
  { "op": "text_equals", "col": 2, "value": "" }
]}
```

and non-blank is {"op": "not", "arg": ...} wrapped around it. Any other column type has no text filter source, so text_equals fails with unsupported ("Column has no text filter source."). Such columns have no empty-string value: blank is {"op": "is_na", "col": N} and non-blank is not applied to it. is_na itself is accepted for every column type.

Numeric values and infinity:

Numeric leaves compare source values without display rounding or an equality tolerance. Extremely small distinct finite values remain distinct. between includes both endpoints and rejects reversed bounds.

A numeric filter operand is a finite JSON number or one of these objects:

```
{"special": "pos_inf"}
```

```
{"special": "neg_inf"}
```

These are the only nonfinite query tokens. Missing/null, NaN, "nan", and "na" queries are rejected. Source negative infinity sorts and compares below every finite value, and source positive infinity above every finite value. Missing and NaN source rows remain false at comparison leaves. Materialized number channels recover infinities from special; null values alone do not distinguish all cases.

Duration:

A column with schema type duration has a units field that is one of secs, mins, hours, days, or weeks. The supported operators are eq, ne, lt, le, gt, ge, between, and is_na; text, in, and Boolean operators fail with unsupported.

A duration operand is either:

- a finite JSON number (or an infinity token as described above) in the column's own units, so 5 on a column in mins means five minutes; or
- a string [-]H:MM:SS[.fff]: an optional minus sign, one or more digits of hours, two digits each of minutes and seconds (00-59), and an optional decimal fraction of seconds. It is converted to seconds and then to the column's units. Anything else, such as 1:5:00, 1:60:00, 1:00, an ISO date, or a string with a sign other than - or with whitespace, is invalid_request. No date or timezone is involved.

Units are converted by these factors: 1 for secs, 60 for mins, 3600 for hours, 86400 for days, and 604800 for weeks (a week is 7 days). Comparisons are between numeric values in the column's units, without any display rounding: the fractional seconds that the display text shows are not used, so a value displayed as 0:00:01 can be larger than a 0:00:01 operand. A clock-time operand that is not exactly representable in the column's units, such as one second in mins, is subject to ordinary floating-point rounding. As for numbers, NA and NaN never match a comparison, and infinities are restored from special.

Date and datetime:

Date operands are strings of exactly YYYY-MM-DD. Datetime operands require YYYY-MM-DDTHH:MM:SS, optionally followed by decimal fractional seconds and either Z or an explicit +HH:MM/-HH:MM offset. With no offset they use the column's timezone, or the local system timezone when the column has no explicit one. Explicit offsets identify instants independently of that local wall time.

Invalid calendar dates are rejected. Hours must be 00-23, minutes/seconds 00-59; offset hours/minutes must be within 00-23/00-59. Whitespace, alternate layouts, and trailing content are invalid. Comparisons use canonical day/second values, independently of producer-defined display strings.

int64:

int64 operands must be canonical decimal strings matching 0 or -?[1-9][0-9]*, in the finite range [-9223372036854775807, 9223372036854775807]. Numeric operands, plus signs, leading zeros, -0, whitespace, exponent notation, missing values, overflow, and the reserved NA value -9223372036854775808 are invalid.

eq and inclusive between compare exact integer values, retaining distinctions beyond 2^{53} . Other numeric operators are unsupported for int64. Reversed bounds are invalid, and missing rows compare false.

Stable multi-column sort:

Terms are applied in array order, with independent ascending/descending flags. If a pair of keys is equal, including two missing keys, comparison continues to the next term. If every term ties, original row number breaks the tie in ascending order. Missing values, including NaN, are equivalent missing keys and come last in both directions. The source missingness mask overrides any finite key produced for a missing source value; missing keys themselves also sort last.

Protocol column type	Comparison keys
boolean, integer, number, duration	Numeric values, including restored infinities.
string, collation C	UTF-8 byte order.
string, collation locale	The producer's selected locale order.
categorical	Level order as declared by the schema.
date, datetime	Canonical days/seconds.
int64	Exact integer order.
Other sortable types	Producer-defined order.

Missing keys sort last in both directions. A producer applies the requested terms in array order and uses original source-row position as the final ascending tie-breaker.

Page results and channel errors:

The grid/rows protocol result has these fields. Private producer representations may use different field names.

Field	Meaning
total	Number of rows matching the filter.
total_unfiltered	Original row count.
offset	Validated requested offset.
limit	Effective row limit after clamping.
data.source_row	Zero-based original source positions for this page.
data.row_labels	Selected explicit row-label strings, or null when absent.
data.columns	Selected columns in requested order; each contains schema, data, and errors.

Page column data requests display, missingness, and values only; it does not contain sort keys or full-column filter text. Display is always character text, with missing display results normalized to

"NA". Missingness and semantic values remain independent of display. For a number or duration column, `special` lists the response-local zero-based positions of NaN, positive infinity, and negative infinity by kind. `na_rows` independently lists missing positions (including NaN); `[]` means no missing rows and omission means extraction failed. These sparse channels use the same format in Snapshot data and pages, and positions are rebased after sort, filter, and slicing. Columns without a plain values representation can return display and missingness without values.

An error object contains a stable code, a human-readable message, and optional details:

Code	Typical cause
<code>invalid_request</code>	Malformed objects, selectors, directions, operands, or reversed bounds.
<code>unsupported</code>	A valid request uses a column capability/type/operator combination that is unavailable.
<code>limit_exceeded</code>	Registration, page width, sort width, AST, or string budget is exceeded.
<code>schema_changed</code>	The producer detects that the active schema or a required channel no longer matches the view descriptor.
<code>producer_error</code>	The producer fails while accessing data or computing a requested operation.
<code>view_not_found</code>	The view id is missing or closed.

R producer extension

Query diagnostics and R conditions:

The R producer can downgrade individual page channels and include channel errors in the response, using `na_rows` for missingness failures. A display failure can yield `<class>` fallback text. Sort/filter requests that require an unavailable full-column channel raise a query error whose details can include the extraction's schema, data, and errors. Failed channel results are not cached, and extraction does not overwrite the registered holder schema. The R package exposes query failures as conditions inheriting from `jdv_query_error`, `jdv_error_<code>`, `jdv_error`, `error`, and `condition` (see [jdv_errors](#)); these classes are not wire fields.

Source slicing contract:

For a data frame, jdv reads each column with `x[[j]]` and selects rows by applying that column's `[]` method. It never calls table-level `[]`, including `data.table`, `tbl_df`, or `sf`. For a matrix it selects a column and rows with `x[rows, j]`. Extraction does not call `base::unclass()` or coerce a source column before its own subset method runs. Filtering and sorting use each full column first; only after the original row indices are fixed does paging slice the requested rows. Classes that need table-level behavior require a future source adapter.

Paged registration retains the source object and a fixed initial schema. A request checks a lightweight fingerprint of dimensions, column names, and column classes. It cannot detect value-only mutation through a shared reference; callers must register again after changing values. Re-registration builds a new holder and cache before replacing the current generation.

Internal entry points:

Function	Result and purpose
<code>paged_table(x, collation = "C", limits = list())</code>	A <code>jdv_paged_table</code> object
<code>query_indices(holder, sort = list(), filter = NULL)</code>	Original one-based R row indices
<code>query_page(holder, offset = 0L, limit = 200L, sort = list(), filter = NULL, cols = NULL)</code>	A private R page with columns
<code>query_clear(holder)</code>	Clear all result caches
<code>query_dispose(holder)</code>	Clear caches and release resources
<code>snapshot_table(x, collation = "C", limits = list())</code>	Eagerly materialized column

```
snapshot_indices(snapshot, sort = list(), filter = NULL)
```

Reference row selection

These are private R representations. `query_page()` retains one-based `data$row_index` and `data$row_names`; Snapshot channel data uses `sortKey`, and R schemas retain class plus R type factor. At JSON encoding, the producer converts these to the protocol forms `source_row` (zero-based), `row_labels`, `sort_key`, `native_type`, and type categorical. The wire conversion does not change the private query result contract.

For example, this internal R request sorts column zero in descending order and returns the first twenty rows, displaying columns zero and two:

```
holder <- jdv:::paged_table(x)
page <- jdv:::query_page(
  holder,
  offset = 0L,
  limit = 20L,
  sort = list(list(col = 0L, desc = TRUE)),
  cols = c(0L, 2L)
)
```

`sort` is an unnamed list of objects containing exactly `col` and `desc`. `desc` is a required, scalar, nonmissing Boolean. A scalar integer-valued plain R double is a valid selector, as is an R integer; Boolean, complex, classed, nonfinite, missing, fractional, or out-of-range selectors are invalid. Sort columns and page columns must not repeat.

The holder's source, initial schema, and collation belong to one fixed table version. Rebuild the holder when source, schema, or the owning registry's generation changes. `query_clear()` alone does not change any of them. The holder itself does not assign table IDs or manage registry generations.

Bounded registration and requested computation:

Registration probes the first 32 rows by default. For a matrix, it slices and probes the first column once and reuses that homogeneous type/schema result for all column positions and names. It retains the original matrix rather than extracting every full column. For a data frame, it probes a finite row slice of each existing column reference, except that attribute-free logical, integer, and double columns whose implicit class dispatches only to the package's default column methods are probed once per type and the result is shared by every such column. Columns with a class, attributes, or any user-defined column method are always probed individually. Probe extraction includes display, missingness, values, and sort keys; it does not compute full-column filter text.

Registration does not create automatic row names or a full `source_row` vector. Without a sort or filter, original row order is generated when requested. Page display extracts actual requested row slices: `x[rows, j]` for a matrix, or a class-preserving slice of a data-frame column. Display is formatted for that slice, independently of full-column filter text.

A filter or sort retrieves only its referenced full source columns and required channels. A text filter requests the full column's text at that point and never substitutes a page's display text. Custom subsetting must preserve the column class and requested row count. Atomic S3 classes without a subset method retain their metadata; classes with row-dependent attributes must supply suitable subsetting behavior.

Three least-recently-used caches have separate entry limits:

Cache	Default entries	Contents
Source column	32	Successfully retrieved full source columns.
Channel	32	Successful full-column channel results, keyed by column and channel.
Index	8	Filtered/sorted row indices, keyed by the validated request.

Hits refresh recency, and insertion evicts the least recent entry beyond the configured count. Independently, each holder keeps its latest successfully computed filtered/sorted row index outside these caches and the `jdv.max_cache_bytes` budget. This lets consecutive pages for one query reuse the index even when the index-cache entry limit or byte budget would evict it. A different query or an unfiltered page releases the latest-index slot; an older result can remain in the LRU index cache if admitted there. `query_clear()`, generation replacement, and view close release both the slot and caches. The retained index can be large, so it adds at most one query result index per open view outside the caches. Cache entry limits are not byte or total-memory guarantees; one column or index can still be large. Every request is validated before an index can be reused.

Failed channel and index computations are not cached. A successfully fetched source column can remain cached after its channel computation fails. A repaired method can therefore succeed on a subsequent channel request without clearing the holder, provided its fixed initial schema permits that operation. A repair that changes a capability rejected by the initial probe requires rebuilding the holder. Probe diagnostics remain attached to the original schema.

jdv's own query code does not change RNG or options. Arbitrary class methods can have side effects, and a query boundary is not a transaction around them.

Validation and default limits:

Object fields are strict: missing required fields, duplicate fields, unknown fields, malformed objects, or unknown operators are errors. NULL filter means no filtering. JSON objects and arrays correspond to ordinary named and unnamed R lists respectively; no request string is evaluated as R code.

Limit key	Default	Meaning
probe_rows	32	Maximum registration probe rows.
page_rows	2000	Maximum returned rows per page; larger requested limits are clamped.
page_columns	256	Maximum selected page columns; excess is an error.
sort_columns	16	Maximum sort terms.
ast_depth	32	Maximum filter depth, counting the root as depth one.
ast_nodes	256	Maximum total filter nodes and maximum factor-membership labels.
text_bytes	4096	Maximum UTF-8 bytes in each string argument.
filter_bytes	65536	Maximum summed UTF-8 bytes in text/date/datetime/decimal/factor-label arguments.
column_cache	32	Source-column cache entries.
channel_cache	32	Full-column channel cache entries.
index_cache	8	Result-index cache entries.

Internal overrides are supplied as a named plain list of recognized keys and positive scalar integers. Their supported range ends at `.Machine$integer.max`; `ast_depth` has an additional ceiling of 128. Operator names are separately bounded to 32 UTF-8 bytes. Factor-label strings, datetime/date

strings, and integer64 decimal strings contribute to the total filter-string budget as well as their individual argument limits.

`offset` and `limit` are nonnegative finite scalar integers, with integer-valued plain doubles accepted. Values above 2^{53} are rejected. `offset` is zero-based within the filtered/sorted result. `limit = 0` and offsets beyond the result produce an empty page; query validation and total-result computation still apply. A larger limit is clamped to `page_rows`.

`cols = NULL` selects every column, preserving source order, and is not subject to `page_columns`. Before extracting page channels, jdv estimates a row count for both implicit and explicit selections that fits the selected schemas, page work budget, and JSON message budget. It preserves every selected column and reduces the effective page limit; if the fixed schema or one row cannot fit, it returns `limit_exceeded`. Final JSON encoding can shorten the page further for unusually large values or custom formatters. Explicit `cols` arrays remain limited by `page_columns` (256 by default). An empty explicit selection returns no data columns while still returning row indices and totals. Selected columns follow selector order. Table width is otherwise bounded by descriptor and response message budgets rather than a fixed registration column count.

R date and datetime parsing:

A valid-looking datetime that the R producer cannot represent in the selected timezone is rejected. Daylight-saving gaps and ambiguities follow the R platform/timezone parser and have no portable normalization guarantee. Fractional seconds use R's representable timestamp precision.

R column mapping (informative):

This section is informative. It describes how the current R implementation (the jdv package) assigns a schema type to a column, and it is not part of the protocol. Which R classes and vectors map to which type can change from release to release without a protocol version change. Clients decide behavior from type, filterable, and sortable, never from the producer-defined `native_type` display strings; see the Column types section of [jdv_spec](#).

R column	Schema type	Notes
Logical vector without a class	boolean	
Integer vector without a class	integer	
Double vector without a class	number	
Character vector without a class	string	
Factor, including ordered factors	categorical	levels and ordered come f
Date	date	
POSIXct and POSIXlt	datetime	tz is the time-zone setting of
<code>base::difftime()</code> , including classes that extend it such as <code>hms</code>	duration	units is the units attribute
<code>difftime</code> whose units attribute is missing or not one of the five known units	other	The units cannot be converte
integer64	int64	
List without a class	list	Not sortable and not filterabl
Matrix or data-frame column	other	Not sortable and not filterabl
Complex and raw vectors	other	Not sortable and not filterabl
Any other S3 or S4 class	other	Displayed through <code>base::fo</code>

The current R producer fills `native_type` with the character vector returned by `base::class()`. This mapping is informative only; clients treat each string as display text and infer no behavior from it. The R producer maps explicit row names to `row_labels` and sends null for automatic row names.

An atomic vector wrapped only by `base::I()` is classified by its contents, so `I(1:3)` is an integer column. An `I()` list is an other column that is sortable and drillable but not filterable. `I()` around a classed vector keeps the classification of that class, so an `I()` factor is categorical column and an `I()` `difftime` is a duration column.

The `drillable` field of the schema is true for unclassified list, `AsIs` list, `vctrs_list_of`, data-frame, and matrix columns. It is false for record columns, `POSIXlt`, other classed lists, S4 objects, and atomic columns.

A custom `integer64` subclass may decorate display through its `format` method; values, ordering, and filters remain exact C operations.

For collation: "locale", the R producer uses `base::xtfrm()` keys under the system locale. Its paged sort passes selected keys to one radix `base::order()` call and uses the per-term decreasing flags with missing values last. It does not use an order permutation as a sort key because that would lose ties. Character C sorting uses the original strings rather than a full-column rank computation.

Reference host

Snapshot reference and shared fixtures:

`snapshot_table()` materializes the column channels, including full filter text. Its reference consumer evaluates filters one row at a time and implements lexicographic scalar comparison with a stable merge sort. It does not call the Paged `base::order()` or vectorized filter evaluator. Request validation, operand parsing, and ASCII folding are shared so accepted requests have one meaning. `Integer64` reference filtering compares canonical decimal strings by sign, digit count, and digits rather than converting them to double.

Snapshot consumers compare supplied sort keys rather than recomputing the producer's native ordering. The `utf8-byte-order` fixture records that `U+E000` sorts before `U+1F600` under C order. For string, categorical, and `integer64` columns, a Snapshot may omit display only when every nonmissing display value equals its semantic value and every missing display value is "NA". The `na_rows` positions reconstruct omitted missing labels. If a formatter uses another missing label, display remains present, including when all rows are missing. Empty comparisons for zero-row columns count as equal.

For supported inputs with valid required channels, Paged and Snapshot must return identical original source-row positions for every sort/filter request. Display strings may differ because R format methods can inspect the whole vector or the requested page when choosing a format.

The version 1 query fixture ships with the package as shared conformance data for query semantics. Its fold entries record ASCII folding. Each cases entry describes typed source columns and queries with expected zero-based positions in original source order. An `r` member is only an R test-harness construction hint, not a wire field. Fixture number special arrays are source-construction hints, not sparse wire channels; they reconstruct nonfinite source values; JSON null means missing data. Factor columns include levels and orderedness; datetime columns include timezone metadata; duration columns include units, and an `r` of `hms` marks a vector with the `c("hms", "difftime")` class structure.

`example-other` uses a synthetic `query_record` class factory: its fixture values array supplies the numeric constructor input, with JSON null becoming R missing data. Its `base::format()` method returns `paste0("value:", as.numeric(x))`, `base::xtfrm()` returns `as.numeric(x)`, and missingness follows `is.na(x)`. The class's runtime values channel is absent (NULL). The JSON fixture's numeric factory input is therefore not an other column's wire values or a numeric-filter

capability. Because other columns are not filterable, the fixture exercises only display, sorting, and `is_na` for it.

Locate it with:

```
system.file("spec", "query-fixtures.json", package = "jdv")
```

Browser viewer

Consuming query capabilities:

A browser viewer builds filter controls from the schema type and the operators defined above. It respects `filterable` and `sortable`, and treats an unknown type as display-only while continuing to use the server for filtering and sorting. It preserves the zero-based `source_row` returned by the producer as row identity; a one-based row label shown to a person is presentation only. Producer-defined `native_type` strings are display hints and do not change query behavior. Snapshot viewers use the supplied sort keys instead of JavaScript's raw UTF-16 string order or an independent browser locale collation. In the `utf8-byte-order` fixture, raw JavaScript comparison could reverse the specified order.

Copyright (c) 2026 jdv authors. Distributed under the repository's MIT license.

See Also

[jdv_spec](#)

jdv_register

Register an object for a transport-independent viewer

Description

Register an object for a transport-independent viewer

Usage

```
jdv_register(
  x,
  title = NULL,
  mode = c("auto", "snapshot", "paged"),
  collation = NULL,
  binding = NULL
)
```

Arguments

<code>x</code>	A data frame or matrix for grid views; an Arrow Table or RecordBatchReader, nanoarrow array stream, or Polars DataFrame for a sequential grid; or an R object to explore in a tree view.
<code>title</code>	Optional title. Reusing a non-NULL title replaces its table generation; NULL titles always create independent tables.

mode	Grid view mode: "auto", "snapshot", or "paged".
collation	Grid string ordering policy, "C" or "locale". When NULL (the default), <code>getOption("jdv.collation", "C")</code> supplies the default. A non-NULL value is kept as the view's refresh intent.
binding	Optional <code>list(name = "x", env = env)</code> naming a variable that holds x, for embedding hosts. The view then follows that variable as <code>jdv_view()</code> does with <code>track = TRUE</code> : when the variable is assigned a different object, the view is rebuilt as its next generation. The variable must currently hold exactly the object passed as x, and it must be readable without evaluation (a plain or already forced binding), or the call fails. Tracking needs R 4.6.0 or later; older versions fail with the unsupported error code. The environment is referenced weakly. Sequential sources cannot be tracked. See the Details of <code>jdv_view()</code> for what is detected. <code>jdv_binding()</code> finds such a binding from a variable name without evaluating it.

Details

Sequential sources are exported through the optional `nanoarrow` package and moved into the viewer without fetching a record batch. Their total row count is unknown. They support forward-only pages, not Snapshot/Paged modes, random access, sorting, or filtering. The mode and collation arguments apply to ordinary R tables. Closing or replacing a view releases its stream and any retained batch. A string call to `jdv_rpc()` cannot consume a sequential page; those pages require the viewer binary transport.

Re-registering a title that is still active reuses its view ID and increments its generation. Closing that view removes the title association, so a later registration starts with a new ID and generation one. Re-register after changing values held by a reference object because the lightweight fingerprint does not detect value-only changes. Paged tables capture `getOption("jdv.max_page_rows", 2000)` at registration; changing the option takes effect after re-registering the view. `jdv.max_work_bytes` and `jdv.max_query_rows` are unlimited by default for full sort/filter work; `jdv.max_cache_bytes` defaults to 32 MiB and `jdv.snapshot_max_bytes` to 10 MiB. Snapshot construction, registration probes, pages, and JSON encoding retain separate finite derived budgets. Each Paged view also retains its latest filtered/sorted row index outside the cache byte budget, so pages for that query do not repeat its sort/filter work. A new query replaces it; an unfiltered page releases the latest-index slot. Older results may remain in the LRU index cache. Clearing, generation replacement, or closing the view releases both the slot and caches. The latest index adds one potentially large index per open Paged view, while the 32 MiB cache budget continues to govern the LRU source-column, channel, and result-index caches. Auto mode also uses `jdv.snapshot_max_cells` (default 10,000) and caps the snapshot byte estimate by `jdv.max_message_bytes`; if Snapshot-only row-name metadata or snapshot construction exceeds its budget, Auto retries registration as Paged. Page, work, cache, and query-row settings are captured when a table is registered. Snapshot and page work allowances are each the smaller of `INT_MAX`, six times their output allowance plus 4 MiB, and a finite `jdv.max_work_bytes` setting. Snapshot uses the smaller of `jdv.snapshot_max_bytes` and `jdv.max_message_bytes`; page work uses `jdv.max_message_bytes`. When `jdv.max_work_bytes` is finite, query work, page extraction, and response encoding share that per-operation allowance. In the event-loop request path, Interrupt behavior depends on R's suspension state when a transport callback begins; if interrupts were already suspended, a long sort or filter may not stop promptly. The viewer UI should show that R is busy. These estimates do not bound total process memory, source storage, arbitrary user method allocations, R allocator and GC overhead, or transient R-internal string conversion and sorting scratch.

Value

A list with the view id and its generation. The view/open message, and any view/closed messages emitted when the registry evicts or replaces older views, are queued as notifications. With an embedded host, take them with `jdv_notifications()`; see `jdv_spec` for the delivery order. If the notification queue overflows, the session ends, every view is closed, and the call fails.

jdv_rpc

Process a viewer request or poll the local transport

Description

With no argument or NULL, explicitly poll the active transport and process at most the configured drain batch. This recovery path bypasses the event gate and does not run reentrantly. The same call checks the variables that tracked views follow and rebuilds the changed views once (see `jdv_view()`), with or without a transport. An embedding host that takes notifications with `jdv_notifications()` calls `jdv_rpc(NULL)` and then `jdv_notifications()` from its own task callback: R runs task callbacks in registration order, so a host callback that runs before the one of jdv would otherwise miss the update until the next command. A JSON request string is handled synchronously as an embedding API and remains independent of the transport. Notifications such as wake and cancel return NULL. The response never carries notifications: an embedding host takes them with `jdv_notifications()` and delivers them before the response.

Usage

```
jdv_rpc(request = NULL)
```

Arguments

request One JSON request string, or NULL to poll the active viewer.

Value

A JSON response string or NULL for a notification.

Driving requests from a script

The event gate dispatches viewer requests only at top level, when `sys.nframe() == 0`; see `jdv_view()`. A script run with `Rscript` that waits in a loop, for example with `base::Sys.sleep()`, never returns to top level while the loop runs, so no request is processed. Poll explicitly with `jdv_rpc(NULL)` inside the loop instead.

Examples

```
## Not run:
# In a script: keep answering the viewer for 60 seconds.
jdv_view(mtcars)
deadline <- Sys.time() + 60
while (Sys.time() < deadline) {
```

```
jdv_rpc(NULL)
Sys.sleep(0.1)
}

## End(Not run)
```

jdv_spec

jdv JSON Lines Protocol Specification

Description

This document is organized into four parts: the common protocol, the R producer extension, the reference host, and the browser viewer. The common protocol describes wire behavior without requiring R. The remaining parts identify behavior specific to the current R producer, the reference host, and the browser client.

Details

The envelope is JSON-RPC-like, but it is not JSON-RPC 2.0: messages use a method and params, while successful responses use id, result, and warnings. A request that expects a response has an id. Notifications such as wake, cancel, and ack omit the id and have no response.

Common protocol

Compatibility and encoding:

- Each JSON transport message is one UTF-8 JSON value framed as one complete JSON value per line with a trailing LF. A partial line is retained only up to the configured request byte limit. An Arrow IPC response may be followed by an opaque binary body as described below.
- The only supported envelope encoding is json; the only supported compression is none. Both fields may be omitted to select those defaults. Unknown values produce an unsupported error.
- A view id is a string. Column positions, source_row values, and sequential first_row offsets are zero-based. source_row is the original source-row position, including after filtering and sorting. Browser row labels display that position plus one.
- Request ids are either strings of at most 128 UTF-8 bytes or JSON integer literal tokens from 0 through $2^{53} - 1$. Fractional and exponent-form numeric ids are rejected, even when their mathematical value is integral. The same rule applies to cancel.params.id. When a response echoes a valid numeric id, its JSON encoder may use exponent notation as long as the exact safe-integer value is preserved.
- Envelope and method-parameter objects ignore unknown fields for forward compatibility. Filter, sort, and stream/next parameter objects remain strict. Duplicate object keys are rejected. Unknown methods return method_not_found.
- The page request method is grid/rows and identifies its owner with params.view. Lifecycle methods use the view names listed below; the former table/open, table/closed, table/schema, and table/close names are not aliases.

- Unknown optional schema fields must be ignored by clients. Protocol v1 does not define cell inspection.
- The parser rejects malformed JSON number grammar, invalid UTF-8, decoded U+0000, trailing non-whitespace data, nesting deeper than 128, more than 100,000 JSON nodes, or an object with more than 1,024 members. The literal six-character content `\u0000` is valid when represented in JSON as `"\\u0000"`; a JSON escape that decodes to U+0000 is rejected. These structural limits apply to inbound requests.

Message envelopes:

A view descriptor is sent as a notification:

```
{
  "method": "view/open",
  "params": {
    "view": "v-1",
    "generation": 1,
    "title": "example",
    "reason": "register",
    "kind": "grid",
    "access": "random",
    "encoding": "jdv-json",
    "capabilities": {
      "seek": true,
      "sort": true,
      "filter": true,
      "known_total": true,
      "next": false,
      "children": false,
      "refresh": true,
      "track": false
    },
    "collation": "C",
    "nrow": 100,
    "ncol": 2,
    "columns": []
  }
}
```

The example omits schema entries for brevity. `view/open` is a producer-to-viewer notification with no request id. `kind` is `grid`; `access` is `snapshot` or `random`; and `encoding` is `jdv-json`. The `capabilities` object and any known capability fields may be omitted; an omitted object or field means `false`. A `grid` descriptor declares the operations it supports. Each column's `sortable` field constrains sorting, and `filterable` constrains value-based filter operators. `Grid children` is `true` if and only if at least one column has `drillable: true`. `refresh` tells the host that `view/refresh` is accepted.

Descriptor field	Grid
<code>view</code> , <code>generation</code> , <code>kind</code> , <code>access</code> , <code>encoding</code>	Required

capabilities	Optional; missing flags are false
title, reason	Optional
nrow, ncol, columns	Required
collation	Required for Snapshot and Random access
data	Required for snapshot access; otherwise omitted

view is a non-empty string and generation is a positive safe integer that identifies the current build of that view. collation is exactly "C" or "locale" and selects the string order described under Stable multi-column sort in the [query specification](#).

reason is a fixed token that tells the host why a view/open was sent. More values may be added; a host treats an unknown or missing value as "register".

reason	Sent when
"register"	A producer opens or replaces a view.
"child"	A child view is opened by object/open.
"tracked"	A producer rebuilds a tracked view automatically.
"refresh"	A view is rebuilt by view/refresh.

A host can bring a view to the front for "register" and "child", and update it quietly for "tracked" and "refresh".

Grid-level flags declare the operations available in protocol v1; each column's sortable field constrains sorting, and filterable constrains value-based filter operators. filterable is false for columns of type list and other, and is_na remains valid when a valid missingness mask is available, even when filterable is false. Snapshot access may use nrow: null when its row count is unknown. Random access requires a non-negative integer nrow, capabilities.seek: true, and capabilities.known_total: true; this lets clients address any requested offset and report the complete row count. Sequential access uses nrow: null and is specified separately below. The protocol does not define total_rows. The descriptor's encoding field does not change the envelope's encoding: "json" setting. Snapshot access's params.data contains the full data payload as described under the Column payloads section below.

Tree descriptors and navigation are an R producer extension described below.

A request envelope has this shape:

```
{
  "id": 1,
  "method": "grid/rows",
  "params": {
    "view": "v-1",
    "generation": 1,
    "offset": 0,
    "limit": 200
  },
  "encoding": "json",
  "compression": "none",
  "deadline": 1791849600.0
}
```

Only method is required at the envelope level. `id` is required for methods that return a result. `params` is required by view methods and `cancel`. Unknown envelope and method-parameter fields are ignored. `encoding`, `compression`, and `deadline` are optional. `deadline`, when provided, must be a finite JSON number containing an absolute Unix timestamp in seconds. A synchronous or queued request that is past its deadline before execution begins receives an expired error response. The configured request byte limit applies to the entire JSON text. The parser also validates strict JSON number syntax and rejects duplicate object keys.

Warnings are collected up to 100 messages, each truncated to 512 characters. If more warnings occur, the array ends with "Additional warnings were omitted."

A successful response is:

```
{
  "id": 1,
  "result": {},
  "warnings": []
}
```

Every response includes a `warnings` array, including an empty array when no warnings were raised. An error response is:

```
{
  "id": 1,
  "error": {
    "code": "invalid_request",
    "message": "Invalid request."
  },
  "warnings": []
}
```

Errors raised after request processing begins are returned to the request caller. Their code distinguishes the failure class; message is bounded text and must not be used as a stable machine-readable discriminator. An error may also include details. Every accepted request receives exactly one terminal response. Work cancelled, expired, or stale before it begins receives an error response with code `cancelled`, `expired`, or `stale_generation`; a closed view receives `view_not_found`. The host matches cancellation ids to pending work before execution. `ack` is transport-owned.

Arrow IPC binary responses:

An Arrow IPC body is reserved for a successful, id-bearing response. Its JSONL header has the ordinary response envelope and a `result.transfer` descriptor:

```
{
  "id": "jdv-request-id",
  "result": {
    "view": "v-1",
    "generation": 1,
    "transfer": { "encoding": "arrow-ipc", "byte_length": 1234 }
  },
  "warnings": []
}
```

The header line is followed immediately by exactly `byte_length` opaque bytes, then by the next JSONL message. The receiver must not decode the body as UTF-8, scan it for LF, or call the response handler before all bytes arrive. The descriptor is not valid on notifications, errors, or ack. `byte_length` is a non-negative safe integer. The response `id` is the host-issued request `id` and must be a string. The UTF-8 JSON header is limited to 64 KiB, and the header, its LF, and body together are limited to 10 MiB. The body has an absolute 30-second receive timeout; fragments do not extend it. Invalid lengths, oversized frames, EOF, or timeout close the producer connection and fail pending requests as disconnected. Cancellation, expiry, and stale or unknown response `ids` still drain a valid body before the next JSONL message is read.

There is no checksum or delimiter after the raw body. A sender that declares a shorter length may be detected when the remaining bytes cannot form a valid JSONL frame, but framing alone cannot detect a remainder that happens to be a valid JSONL message.

The host forwards a response to the browser as one WebSocket binary message: four bytes containing the UTF-8 JSON header length as an unsigned little-endian integer, followed by the JSON header, followed by the unchanged raw body. The header is rewritten with the browser request `id` and session. The rewritten header remains limited to 64 KiB, and the complete WebSocket message to 10 MiB. If either limit or the request's `max_bytes` is exceeded, the host returns one `limit_exceeded` error after draining the producer body and continues with the next JSONL frame.

For a request that can return a binary body, `params.max_bytes` is optional. When present it must be a non-negative safe integer; zero is valid. Its default is the effective binary body cap. The producer caps a response at the smaller of `params.max_bytes` and its configured message budget after reserving space for the JSON header and framing. The host applies the producer-reported cap and checks the returned byte length against the request before forwarding it. After draining the body, the host checks that the response view and generation match the pending request; a mismatch returns one `invalid_request` or `stale_generation` terminal error.

Sequential Arrow views:

Sequential descriptors have `kind`: "grid", `access`: "sequential", `encoding`: "arrow-ipc", `nrow`: null, and a non-negative integer `ncol`. The common view, generation, and columns fields are also required. Only the next capability is true; seek, sort, filter, `known_total`, children, refresh, and track are false. The producer supplies the schema and owns the forward-only Arrow stream for the lifetime of the view generation.

Each Arrow IPC body carries its logical schema. The descriptor's columns array remains in original source order and retains each column's index, name, type, unsupported, `ipc_index`, position, sortable, filterable, and drillable fields. `index`, `position`, and non-null `ipc_index` values are zero-based. `unsupported` is null for supported columns and otherwise explains why a column was omitted from IPC; its `ipc_index` is null. Names may repeat and must not be used to identify IPC vectors; use `ipc_index`. Even a page with no supported columns reports its row count in page metadata.

`stream/next` accepts only view, generation, `max_rows`, and `max_bytes`. `max_page_rows` is a required descriptor field containing the positive safe-integer row cap captured when the view is opened. `max_rows` defaults to `min(200, captured_cap)` and must be a positive safe integer no larger than that cap. The viewer requests `min(200, descriptor.max_page_rows)` rows and validates returned row counts against that request. The host rejects an invalid or missing `max_page_rows`, and rejects `max_rows` above the captured cap before granting a lease. `max_bytes` uses the shared binary response budget above. Zero is a valid budget but returns `limit_exceeded` before fetching because it cannot hold an IPC stream. Offset, sorting, filtering, and count requests are rejected. Random grid/rows is unavailable for sequential views.

A successful binary response carries view, generation, first_row, rows_fetched, rows_seen, done, and the reserved transfer object. first_row is the zero-based source cursor before this page; rows_seen = first_row + rows_fetched. Each page is a self-contained IPC stream and never crosses an input batch boundary. done becomes true only after the producer's end-of-stream sentinel has been observed. A data page can therefore have done: false even when it contains the last rows. A following schema-only zero-row page can report done: true. Empty batches are skipped with bounded work; reaching that bound yields zero rows and done: false, which clients must not treat as EOF.

The host leases a view generation to the first attached browser whose next request is accepted. Other browsers receive in_use while that owner remains attached, including between pages. At most one next request is pending per view generation. Cancellation, timeout, or owner detachment does not fail the producer stream and does not rewind a sent request. Its slot remains reserved until the late terminal response is drained and discarded. Detachment releases the lease; a new owner then starts at the current source cursor. No additional host-to-producer failure control is sent. The host validates current generation and budgets before granting a lease. A producer connection loss closes all its views.

Producer schema/array/next failures and a single row exceeding the page budget make the producer stream terminally failed and release its resources. Later next requests return stream_failed. Ordinary encoder/allocation failures retain the carry and uncommitted offset for retry. Request cancellation and host timeout are separate from these producer failures.

Methods:

Method	Direction	Kind	Result
view/open	Producer to viewer	Notification	Descriptor and, for snapshot access, all column payloads.
view/schema	Host to producer	Request	Current dimensions and schema.
stream/next	Host to producer	Request	One forward-only Arrow IPC page and cursor metadata.
grid/rows	Host to producer	Request	One selected page from a random-access grid.
view/refresh	Host to producer	Request	Rebuilds a view as the next generation of the same id.
view/close	Host to producer	Request	Closes one view generation.
view/closed	Producer to host	Notification	Removes a closed or replaced generation from the host cache.
hello	Producer to host	Notification	Announces protocol and producer metadata for a new connection.
wake	Host to producer	Notification	No result; asks the producer to check for pending work.
cancel	Host to producer	Notification	Names pending work for the producer to cancel.
ack	Producer to host	Notification	Cumulatively acknowledges complete host-to-producer JSONL frames

view/schema:

Parameters are view and generation. The result contains view, generation, nrow, ncol, and columns. A generation that is no longer current produces stale_generation when handled synchronously. A missing or closed view produces view_not_found.

view/refresh:

view/refresh is a request with an id. Its params identify the active view using the required view and generation fields. The descriptor's capabilities.refresh field gates this operation: when it is false or omitted, the host must not offer refresh, and a request that nevertheless reaches the producer fails with unsupported. A missing view fails with view_not_found; a generation that is no longer current fails with stale_generation.

On success, the producer replaces the descriptor under the same view id and advances its generation by exactly one. It sends the replacement view/open notification before the successful response. The result contains the view id, previous_generation, the new generation, and refreshed: true. The view/open is authoritative for the active descriptor; the response confirms the operation and must not restore an older generation in the host or viewer. A failed refresh returns an error without publishing a new generation. The replaced generation does not need a separate view/closed notification.

view/close and view/closed:

view/close is a request with an id. Its params identify the active view using the required view and generation fields. A missing view fails with view_not_found; a generation that is no longer current fails with stale_generation. On success, the producer removes that generation and returns a result containing the same view and generation with closed: true. The response completes an explicit host-initiated close; the producer does not also send view/closed for that close.

A producer sends view/closed as a notification when a view generation becomes unavailable without a matching host-initiated view/close request or a successor view/open notification replacing it. Its params contain view and generation; an optional reason may provide additional producer-defined context. A host removes the cached view only when both identifiers still match its active descriptor. It ignores a notification for a generation that has already been replaced or removed.

grid/rows:

Parameters are:

Field	Required	Meaning
view	Yes	Registered view id.
generation	Yes	Generation returned by view/open.
offset	No	Zero-based offset after filtering and sorting; defaults to 0.
limit	No	Requested page size; defaults to 200 and is clamped to the producer's page row limit.
sort	No	Array of { "col": zeroBasedPosition, "desc": boolean } objects; defaults to no sort.
filter	No	Filter AST defined by query specification ; defaults to no filter.
cols	No	Array of zero-based column positions; omitted or null selects all columns.

The result contains view, generation, total, total_unfiltered, offset, limit, and data. The returned offset equals the requested offset, and the returned limit is a safe integer between the number of returned rows and the requested limit. data contains source_row, optional row_labels, and columns. The effective limit is clamped to the query engine's row limit and may be reduced further to fit the JSON output budget. For an empty page, source_row and explicit row_labels are empty arrays; columns still contains one schema/data/errors wrapper per selected column, with empty channel arrays. Only cols: [] produces an empty columns array.

Column selectors are positions rather than names, so duplicate or unusual column names are safe. cols: [] requests only the row metadata. An omitted or null cols selects every column. A producer may reduce the effective limit to satisfy its response budget; if the fixed page schema or one row cannot fit, the request returns limit_exceeded. The R producer estimates a row count for the selected columns against its page work and response budgets before extracting page channels, preserving all selected columns. Snapshot access rejects row requests with unsupported because its payload is host-cached.

hello:

At the start of each producer-to-host connection, the producer sends a hello notification with no request id:

```
{
  "method": "hello",
  "params": {
    "protocol": 1,
    "session": "producer-1",
    "producer": {
      "name": "example-producer",
      "version": "1.0.0"
    },
    "wake": true,
    "limits": {
      "max_unacked_frames": 264,
      "max_unacked_bytes": 1048577,
      "queue_messages": 256,
      "queue_bytes": 16777216,
      "max_request_bytes": 1048576,
      "max_message_bytes": 10485760
    }
  }
}
```

session is an opaque session identifier. It is 1–128 ASCII bytes and may contain only letters, digits, ., _, :, and -. A receiver must not infer a process id or connection counter from it. A host rejects an invalid identifier, and rejects a duplicate active identifier with in_use. producer identifies the producer: name and version are UTF-8 strings from 1 through 256 bytes each. runtime is optional; when present, it has string name and version fields, each also 1 through 256 UTF-8 bytes. A receiver rejects the handshake if any supplied metadata value is empty, invalid UTF-8, or exceeds the byte limit. Protocol major version 1 is the v1 contract in this document. A peer that does not support the announced major version rejects the handshake with unsupported; it must not continue with incompatible message semantics.

wake is optional and, when present, is a Boolean. An omitted value means false; any other type is invalid. When true, the host must send periodic wake notifications while requests await terminal responses. When omitted or false, the host must not send wake. A producer that needs periodic wakes sets wake to true.

hello is the first producer message on a new connection. A disconnect closes the views associated with that connection.

params.limits is required and contains all six required limits, each a positive safe integer: max_unacked_frames and max_unacked_bytes bound the complete host-to-producer JSONL frames and bytes sent without cumulative acknowledgement; queue_messages and queue_bytes bound producer-side pending work; max_request_bytes and max_message_bytes bound one request and one producer message. A host uses the smaller of each reported value and its own configured bound. Missing or invalid limits reject the handshake.

wake and cancel:

wake has no response and no parameters. A host must not send it when `hello.params.wake` is omitted or false; when that value is true, the host must send wakes repeatedly at a regular interval while one or more requests are awaiting terminal responses. A producer that needs periodic wakes sets `wake` to true. A host may choose any regular interval and must keep wakes within applicable frame, byte, and send-rate limits.

cancel has no response and takes `{ "id": requestId }` in `params`. It identifies a request by the same string-or-safe-integer rules as envelope `ids`. `Ids` retain their JSON type, so the string "1" does not match numeric id 1. A producer cancels matching work that has not begun and sends one cancelled terminal error. Work already running is not forcibly interrupted by this notification and produces its ordinary terminal response. The producer validates the full control envelope, including encoding, compression, and deadline, before it changes pending work. Unsupported encoding or compression and an expired recognized cancel have no effect and produce no notification response. Malformed JSON, an invalid raw id token, or an invalid deadline type can be rejected during request validation with a best-effort `invalid_request` response whose `id` is null; these errors do not change pending work.

ack:

Every producer must send `{ "method": "ack", "params": { "frames": N } }` after it has read one or more complete frames. `N` is the cumulative number of complete host-to-producer JSONL frames read on that connection, including wake, cancel, malformed, and rejected frames. It is a safe integer and never decreases. The host releases frame and byte credit through that cumulative frame boundary.

Column types:

The schema type of a column is one of the following values. This table and the rules below are the protocol contract; filter operators and operand forms are defined in the [query specification](#).

type	Additional schema fields	Value filter operators
boolean		<code>is_true</code> , <code>is_false</code>
integer, number		<code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> , <code>between</code>
string		<code>contains</code> , <code>starts_with</code> , <code>ends_with</code> , <code>text_equals</code>
categorical	<code>levels</code> , <code>ordered</code>	The text operators, and <code>in</code>
date		<code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> , <code>between</code>
datetime	<code>tz</code>	<code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> , <code>between</code>
int64		<code>eq</code> , <code>between</code>
duration	<code>units</code>	<code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> , <code>between</code>
list, other		None

Every column type also accepts `is_na` when a valid missingness mask is available. A client decides what it can do with a column from `type`, `filterable`, `sortable`, and the grid capabilities alone. The schema's optional `native_type` field is a producer-defined array of display strings. A client may show those strings as a hint but must not infer semantics from them. How R vectors map to protocol types is described, as informative text, in the R column mapping section of [query specification](#), and can change without a protocol version change.

A client that receives a type it does not know must treat the column as display-only, exactly like other: it follows the column's `sortable` for sorting, offers no value-based filter, and uses only `is_na`. It must not fail, hide the column, or guess from `native_type`. Because of this rule, a new

type can be added without changing the protocol major version. For the same reason, a later release may add operators or operand forms to a known type; a client offers only the operators it knows for that type and must be prepared for the server to answer a request with an unsupported error, which is an ordinary failure of that request and not a protocol violation.

Column payloads:

Column identity is its zero-based position; name is display metadata and may be empty, duplicated, or otherwise unusual. A schema entry includes position, name, type, sortable, and filterable; native_type is optional and, when present, is a producer-defined array of display strings. Categorical schemas also include levels and ordered; datetime schemas include tz; duration schemas include units. Unknown optional schema fields must be ignored by clients so future schema extensions remain compatible; the Column types section defines how to treat an unknown type value.

A column payload is columnar. Channels may be omitted when unavailable:

```
{
  "values": [1, null, null],
  "display": ["1", "NaN", "NA"],
  "special": {"nan": [1]},
  "na_rows": [1, 2]
}
```

values, display, and sort_key, when present, have one entry for each row in the payload. na_rows lists missing row positions in this response, starting at zero, in strictly increasing order without duplicates. Positions must be integers within the payload's row count; they are not source_row identifiers. na_rows is sent whenever missingness extraction succeeds, including [] when no rows are missing. Omission means missingness is unavailable. It is authoritative; clients must not infer missingness from display text. A producer may omit an unavailable channel while keeping other channels usable. Clients use descriptor capabilities and channel availability to decide which operations are available.

special, when present, is an object with only nan, pos_inf, and neg_inf keys, in that producer order. Each value is a nonempty array of row positions with the same integer, order, uniqueness, and range requirements as na_rows. Empty kinds are omitted, and the whole field is omitted when no kind occurs. Kinds must not overlap. NaN positions also occur in na_rows when missingness is available, following R's is.na() semantics; infinities are not missing. Clients reject invalid position channels as schema_changed. The old na field and per-row special arrays are not part of this format.

In snapshot access, data.columns entries contain the complete channel data. A random-access response wraps each selected column as { "schema": ..., "data": ..., "errors": ... }, preserving the page-time schema. The data object inside that wrapper uses the same channel format. Random-access rows do not include sort_key; sorting remains on the producer side.

The following encoding rules apply:

- boolean, integer, and finite number values use JSON booleans or numbers. duration values are numbers in the schema's units (secs, mins, hours, days, or weeks), with the same missing and non-finite rules as number; display carries the formatted text. Missing values use JSON null and are identified by na_rows.

- Non-finite doubles use JSON null in values; special identifies NaN and positive or negative infinity by row position. NA is represented only by na_rows. Numeric comparisons and sorting restore infinities from special; na_rows controls missingness and NA-last placement.
- int64 values are canonical base-10 JSON strings, never JSON numbers, to preserve exact integer precision. na_rows identifies missing entries. The display channel may be omitted only when its nonmissing labels equal values and every missing label is "NA". The client reconstructs display from values, using "NA" for missing entries. A producer may retain display when its display text does not meet this rule.
- string and categorical values are strings. The display channel may be omitted under the same equality and missing-label rule. Categorical order and labels come from levels and ordered in the schema.
- date values are ISO YYYY-MM-DD strings. datetime values are ISO timestamp strings; the schema's tz records the source time-zone setting.
- In snapshot access, sort_key is present when a value's ordering cannot be recovered from its values and special entries. It carries the producer's ordering key; consumers compare keys rather than recomputing a producer's native ordering. A null sort key is ordered after every non-null key, regardless of na_rows. random-access rows never include sort keys.
- source_row always reports zero-based positions in original source order. Snapshot data and random-access page data may contain one row_labels string per row; an absent or null field means there are no explicit row labels. Grid descriptors do not contain row_labels.

Snapshot-access payloads contain all table rows in one data object. Random-access payloads contain only the requested row slice. The protocol guarantees matching sort and filter semantics, not identical display strings.

Filter and sort request shape:

The filter is the neutral AST defined by [query specification](#), not a UI library's filter model. Objects contain exactly the fields required by their operator. Sort is an array of objects with exactly col and desc; desc is a required scalar Boolean. Duplicate sort and page columns are invalid. Cross-consumer examples of filters, values, and expected original row indices are in the fixture described in the [query specification](#).

Numeric filter special values use an explicit object, never a non-JSON number:

```
{ "op": "eq", "col": 0, "value": { "special": "pos_inf" } }
```

int64 filter values are canonical decimal strings. Missing and special value semantics follow the [query specification](#).

Errors and limits:

Error codes are stable machine-readable values. The error object contains code and message, with optional details. The current codes include:

A consumer treats an unrecognized code as a general producer failure and may display the code without requiring code-specific recovery behavior.

Code	Meaning
invalid_request	Malformed JSON, missing required fields, strict filter/sort field errors, or invalid parameters.
unsupported	A known operation or encoding is not supported for this table or protocol version.

limit_exceeded	A configured request, page, or JSON byte limit was exceeded.
schema_changed	The active view schema no longer matches its descriptor, or a required channel is unavailable.
producer_error	A producer method or other request computation failed.
interrupted	Request processing began and was interrupted.
cancelled	The accepted request was cancelled before processing began.
expired	The request deadline elapsed before processing began.
unavailable	A well-formed object path no longer resolves or cannot be read safely.
stale_generation	The request names a generation that is no longer active.
view_not_found	The view id is missing or closed.
cache_changed	A host-only sequential append cursor differs from the observed tail; refresh the host page catalog before
in_use	A session id is already active, another browser owns the sequential lease, a next request is still pending,
stream_failed	A sequential producer encountered a terminal failure; close or register a new stream to continue.
method_not_found	No implementation exists for the requested method.

Request and response size limits are negotiated in hello. A producer that cannot satisfy a request or configured budget returns `limit_exceeded`. Per-column channel failures may be reported in the page payload without failing the entire request.

R producer extension

object/children:

This method is available only on tree views. A grid with `children: true` uses `object/open` to capture a cell; it does not expose `object/children`.

Parameters are `view`, `generation`, `path`, and optionally `offset`, `limit`, and `attributes`. The `path` is a full typed path from the root value held by the named tree view; an empty path selects that root. `offset` is zero-based and defaults to 0; `limit` defaults to 200 and must be between 1 and 500. `attributes` is a Boolean that defaults to `true`; any other value, including `null`, returns `invalid_request`. A path contains at most 32 steps. Each step is one of:

Step type	Fields	Meaning
index	index	One-based position in an unnamed or positional object.
binding	name	Environment binding name.
slot	name	S4 slot name.
attr	name	Attribute name.

The result contains the same `view`, `generation`, full `path`, and `offset`, plus `children` and `total_children`. Each child has its one-based index, bounded label, string-array `class`, `type`, bounded `summary`, `has_children`, `viewable`, and its full typed path when the child is addressable. `class` and `type` have the same meaning as in a tree descriptor; a child that is unavailable has an empty `class` and no `type`. A child whose binding, slot, or attribute name exceeds 1,000 characters is not addressable: it has `viewable: false`, `has_children: false`, `unavailable: "path_limit"`, and may omit `path`. Other children include the full typed path. When a value cannot safely be read, the child carries a bounded unavailable reason. Child text, each class string, and each summary are at most 1,000 characters. Summaries inspect at most five values or children and at most three levels deep. Environment names are enumerated locally and paged; active and delayed bindings are reported as unavailable without reading their values.

Children are the elements of the object followed by its attributes. Within an atomic vector (logical, integer, double, complex, character, or raw), every element is a leaf with `has_children: false`. Its summary has the form `<class> (1): <value>`, for example `double (1): 1.5`, and always includes the value regardless of depth. A page of elements is read and formatted as one range, so only the requested range of an ALTREP vector is read. Dimensionless atomic vectors with a class, excluding S4 objects, use the column display format for the requested range; if that formatting fails, their underlying atomic values are shown instead. A missing value is shown as NA, and strings and formatted values are cut at 160 characters. A factor element, and every value in a factor's summary, is shown as its level name; a missing code is NA, and a code outside the levels is shown as `<invalid code N>`. An atomic vector has children only if it has more than one element, or if attributes are listed and it has any, so a length-one vector without attributes is a leaf.

A list object with a class-specific format method shows its logical value count as `(N values)` in the summary, using the class's `length` method; one value is written `(1 value)`. The summary may preview up to five formatted values using the class's `subset` method and the column display format; this preview does not change the count. A zero length shows `(0 values)` without calling `subset` or `format`. If `length` fails or returns an invalid value, the summary shows the class without a count. If `subset` or `display` fails, the valid count remains. At the summary depth limit, only the class is shown, with no count, and `length` and `format` are not called. POSIXlt and classed vctrs records use this rule. Plain lists, data frames, and AsIs lists retain their existing summaries. Attributes can change `total_children` without changing this logical value count. This formatting applies only to summaries; listing children does not dispatch list subset methods.

With `attributes: false`, the attribute children (`attr steps`) are left out: of every vector and other object, including a vector's names, which remains visible as the label of each element. `total_children`, `paging`, and `has_children` of the listed children follow the setting, and attributes that are present are neither counted nor paged. Environment bindings and S4 slots are the contents of those objects, not attribute children, so the setting does not hide them.

Tree navigation never evaluates a promise, active binding, classed `[[`, `$`, or `names` method. Paths are resolved again for each object/children request.

object/open:

Parameters are `view`, `generation`, a non-empty path, and optional `mode` and `collation`. `mode` must be exactly `"auto"`, `"snapshot"`, or `"paged"`; when omitted, it defaults to `"auto"`. `collation` must be exactly `"C"` or `"locale"`; when omitted, it uses the same `getOption("jdv.collation", "C")` default as `jdv_register()`. These options apply when the selected value opens as a grid and do not inherit from the parent view. If the selected value opens as a tree, valid `mode` and `collation` values are ignored. Invalid values, including explicit null, return `invalid_request`. `object/children` does not accept these options.

On a tree view, every path step uses the type discriminant from the table above. On a grid view, the first step must select a drillable cell:

```
{
  "view": "v-1",
  "generation": 1,
  "path": [{ "kind": "cell", "source_row": 204, "column": 2 }],
  "mode": "paged",
  "collation": "C"
}
```

source_row is the original, zero-based source-row position, independent of sorting, filtering, and pagination. column is the zero-based schema position, including when column names are duplicated. kind: "cell" is allowed only at the start of a grid path. Remaining steps, if any, use the tree type forms above and are resolved within the captured cell value. The complete path, including its cell step, contains at most 32 steps. Grid object/open requires capabilities.children: true and columns[column].drillable: true; malformed selectors or non-drillable columns return invalid_request.

Each grid column schema includes the Boolean drillable, which says whether the column's cells can be selected by kind: "cell". This metadata is independent of type, sortable, and filterable; clients must not infer drillability from the type, native_type, or display text. The R columns that are drillable are listed in the R column mapping (informative) section of [query specification](#). Snapshot views retain the R source only when a drillable column needs subsequent cell capture. Retained source references are released when the view closes.

object/open resolves the selected value once, holding that captured value in the new view. A path that cannot be resolved returns unavailable; malformed paths return invalid_request, missing views return view_not_found, and old generations return stale_generation. The result is the child's view/open.params descriptor, but the child is registered by the view/open notification that precedes the response (see the Notification delivery section); a host must not create or revive a view from the response, and may use it only to find the child. If building the response fails, the child is closed and no view/open for it is sent. The child descriptor has reason: "child". If the parent has a title, the child's title is an R expression built from the parent's title and the opened path; if the parent has none, the child has none either. A list element uses x\$name when its name is nonempty and unique and x`a b` when the name is not a syntactic name, and x[[3]] (one-based) otherwise. An element of a named atomic vector uses x[["name"]] under the same rule. An S4 slot is x@slot, an environment binding is x\$name, an attribute is attr(x, "names"), and a grid cell is df[12, "col"] with the original zero-based source_row 11, or df[12, 3] with a one-based column position when the column name is missing, empty, or duplicated. A child of a child continues from the child's title, for example x\$a[[2]]@data. The title is truncated to 4,096 bytes. It is a display label only: it is not guaranteed to be valid R, a host must not evaluate it, and it does not identify the view or take a title slot. A tree child starts a new root with an empty path; the descriptor may identify its parent with parent. The limit is 32 live descendant views total per root, counting nested tree and grid views across all levels. Closing a child releases its slot for a later child view. Closing, replacing, or evicting a parent closes its descendants; each cascaded child emits view/closed with reason: "parent_closed".

The browser uses the same child-view navigation for tree and grid parents. Opening controls appear only in drillable cells and use the displayed row's original source_row (shown as a one-based row number). The tree viewer shows the root class, and a checkbox, checked by default, sets the attributes parameter of its object/children requests. View breadcrumbs restore the parent's sort, filter, page, horizontal scroll, or tree path and child-page offset. A pending child open is tied to the active parent view and generation, separately from page query epochs: Snapshot sort, filter, and page operations remain local and do not cancel the captured cell request. Returning to the R prompt lets that original request complete without another browser request. Selecting a different view, replacing its generation, or closing it cancels its pending browser requests. Display text and navigation labels are inserted as DOM text.

R 4.6 and newer use the public experimental R_GetBindingType() and R_getVarEx() APIs to distinguish Value, Forced, Delayed, Unbound, Missing, and Active bindings. Only Value and Forced bindings are read; Delayed, Unbound, Missing, and Active bindings are opaque, and

delayed expressions are never deparsed. On older R versions only active bindings can be identified; other environment bindings are treated as opaque. For ALTREP values, bounded element or region access may call the class's representation callbacks to read values needed for a page or summary. This does not force delayed promises or invoke active bindings. Tree access does not request DATAPTR; the protocol does not guarantee total memory use by an ALTREP implementation. If reading an individual child fails, that child is returned with an unavailable reason and sibling children remain available. S4 .Data is opaque; slots are inspected as stored values.

view/refresh:

The common request, capability gate, generation transition, notification, and response contract is defined above. This section describes how the R producer performs that operation.

The view is rebuilt from the value captured when it was registered, so the rebuild shows the same data unless the captured object is a reference object that changed in place. A tracked view (see Binding tracking) is instead rebuilt from the current value of its variable, read without evaluation; it fails with unavailable when tracking has ended (the reason is in the message) or the variable is a promise or active binding. A view whose tracking is "paused" can be refreshed too: it is rebuilt from the current value of its variable, the new object becomes the baseline, and the state stays "paused". The rebuild happens even if nothing changed and always advances the generation by one. The registration mode and collation are reused as they were requested: an "auto" mode is decided again (a grid may move between Snapshot and Paged, and the effective mode is never frozen), an explicit mode is kept, and a NULL collation intent reads `jdv.collation` again. A refresh of a titled view keeps its title association, so a later registration with that title continues from the refreshed generation.

The replacement is built first. Only if the view, its generation, and (for a child view) its parent are still current is it published, as one step, under the same view id with the new generation, and the new view/open is queued. Then every descendant view is closed, deepest first, with view/closed notifications carrying reason: "parent_closed". If releasing the old generation closes the refreshed view, its view/closed follows the new view/open, so the host never keeps a generation R no longer has. The old generation of the refreshed view itself gets no view/closed; its view/open successor replaces it. As for any request, these notifications precede the response. A refresh that fails, is interrupted, or finds the view stale leaves the generation and all children unchanged. A `jdv_close()` or `jdv_disconnect()` issued by user code while the replacement is being built keeps its effect: the closed view is not revived and the request fails with `view_not_found`. `jdv_register()`, `object/open`, or another refresh attempted by user code during the rebuild fails with `in_use`. If the notification queue overflows, the session ends as described under Notification delivery.

Refreshing a child view keeps it a child of the same parent generation (the descriptor repeats parent); it is refused with `view_not_found` once the parent was refreshed, replaced, or closed. Sequential views are not refreshable (unsupported) because a consumed stream cannot be rewound. Errors are `invalid_request`, `view_not_found`, `stale_generation`, `unsupported`, and the usual `limit_exceeded`, `producer_error`, and `interrupted`.

A hub that forwards requests treats a sent view/refresh specially. The successful refresh queues the new generation's view/open before its own response, and a hub normally discards pending requests of an older generation when it sees a newer view/open. The response of a sent view/refresh is not discarded for that reason, whether it reports success or failure, as long as the request was not timed out, cancelled, or lost to a disconnect. The hub checks that the response matches the request (same id, view, and requested generation as `previous_generation`, and a generation one higher on success). The view/open notifications alone update what is displayed; the response only confirms the operation and must never move the display back to an older generation.

view/track and view/tracking:

```
{ "id": "track-1", "method": "view/track",
  "params": { "view": "v-1", "generation": 3, "enabled": false }}
```

Field	Request	Result
view	Required; the view id	The view id
generation	Required; must be the current generation	The generation after the request; one higher when rebuilt
enabled	Required boolean; false stops, true resumes	Omitted
tracking	Omitted	The tracking object after the request

Other fields are rejected with `invalid_request`, as is a non-boolean `enabled`.

`enabled: false` changes the state from "active" to "paused". The binding and the baseline object are kept, and the variable is not examined while the view is paused. `enabled: true` changes "paused" back to "active". At that moment the current value of the variable is read and compared with the baseline. If it differs, the view is rebuilt in the same way as an automatic update: a new view/open with reason: "tracked", `tracking.state: "active"`, and the next generation is queued, and the result carries the new generation. If it is the same object, the generation does not change. A value that cannot be read without evaluation (a promise or an active binding) is looked at again later, as for any check. If the rebuild fails, a `view/update_failed` is queued as for an automatic update and the view stays active at the same generation. A request for the state the view already has succeeds and changes nothing. If the variable turns out to be gone, the state becomes "lost" (announced as below) and the result reports it.

A view that has no tracking field, or whose state is "lost", fails with `unsupported`, and the message names the reason. A generation that is not current fails with `stale_generation`, and an unknown view with `view_not_found`. The other errors are `invalid_request`, `producer_error`, and `interrupted`.

view/tracking is a notification:

```
{ "method": "view/tracking",
  "params": { "view": "v-1", "generation": 3,
    "tracking": { "name": "df", "state": "lost",
      "reason": "removed",
      "message": "the binding was removed" } } }
```

R queues it when the state changes without a new view/open: when following is lost, and when view/track pauses, or resumes without rebuilding. When view/track resumes and rebuilds, the new view/open carries the state and no view/tracking is sent. `generation` is the generation the state belongs to; a host ignores a notification for a generation it no longer shows, and it updates its record of the view's tracking and capabilities.`track` (false for "lost"). The notification precedes the response of the request that caused it, like every notification. Following is lost when the variable is removed or its environment is garbage collected, which R finds in the task callback, in a view/refresh, or when view/track resumes.

A hub that forwards requests may treat a sent view/track like a sent view/refresh: a successful rebuild raises the generation by one, and the view/open alone updates the display.

view/close and view/closed:

The common request, response, and notification contract is defined above. In R, a successful view/close removes the entry from the active registry immediately. If a request is already using its entry, source and holder cleanup is deferred until that request releases it.

When the registry evicts its oldest active table to satisfy `jdv.max_tables`, it queues a view/closed notification with the evicted view and generation. The notification enters the session queue ahead of the associated new view/open, and the host delivers both in queue order. `jdv_close()` queues view/closed for views it closes. Re-registering an active title replaces the same id with a newer generation; after close or eviction, a later registration uses a new id and generation 1.

Closing a tree view removes its child views recursively. A view/closed notification for a child closed by its parent includes `reason: "parent_closed"`; the explicitly closed root has no such reason. Replacing or evicting a parent has the same descendant-close effect.

R-specific descriptor and runtime details:

The R producer adds `class` and `type` to tree descriptors. `class` is a string array following `base::class()` semantics for the root object, and `type` is the string returned by `base::typeof()` (for example "list", "double", "environment", or "S4"). Its tree descriptors use `kind: "tree"`, `access: "random"`, `encoding: "jdv-json"`, and include a path array that is empty at the root. Grid descriptors omit `class` and `type`.

R may include a tracking object when a view follows a variable. It has the form `{"name": "df", "state": "active"}`: `name` is the variable name and `state` is "active", "paused", or "lost". The lost state includes a reason token and optional message. A view that does not follow a variable omits tracking and sets `capabilities.track` to false. The R producer sends reason on every view/open; it sets the token according to the table in the common descriptor section.

R column channel diagnostics use an empty `errors` array on random-access pages when no channel failed, and a named object of channel-specific messages when errors are present. Snapshot-access columns include `errors` only when a channel failed. Missingness failures use the `na_rows` diagnostic key, matching the wire channel name. Channel failures are reflected in the schema capability flags; failed channel results are not cached.

The R package uses base R and its bundled cJSON implementation at runtime; it does not require jsonlite. Consumers can parse and construct JSON with their own tooling and pass a JSON string to `jdv_rpc()`. Inbound requests follow the common parser limits above. Generated R messages are bounded by the outbound byte limit and are not revalidated with the request parser's node or object-member limits.

The R producer generates session ids as `r-<pid>-<connection counter>` and does not use or change R's random state. It identifies itself as jdv running on R and sets `wake: true`. Its grid descriptors send all eight known capability flags. The R API emits hello on explicit connect or with the first registered view on an implicit connection; a disconnect releases the process-wide view registry.

In the R local transport, wake checks whether queued requests can run, but a wake does not dispatch work while R is in a function frame or at a receive-only API preflight. A 50 ms interval is a valid host choice but is not required by the protocol. A running R method is not forcibly interrupted by cancel.

The cJSON writer first tries a 15-significant-digit representation for a finite double and accepts it only when its own read-back is exactly equal to the source; otherwise it falls back to 17 significant

digits. The emitted text round-trips to the source double with correctly rounded parsers. Negative zero is written as 0, without its sign.

The nanoarrow adapter obtains the schema during registration without advancing the stream, then transfers stream ownership to the native layer. Close, replacement, eviction, parent cascade, namespace unload, and producer disconnection release the stream and any carried batch exactly once. A release callback may execute R code; registry identity and generation are checked again before publishing a result. Releases requested inside an active producer callback are deferred until that callback finishes. The synchronous `jdv_rpc()` API rejects `stream/next` before consuming the producer; the local transport uses its private binary-capable entry point.

R producer budgets, page retries, and interruption behavior are described in the following implementation notes.

Per-column channel failures are represented in the page payload and do not necessarily fail the whole request. The request and response byte limits are `jdv.max_request_bytes` (default 1 MiB, minimum 128 bytes) and `jdv.max_message_bytes` (default 10 MiB, minimum 1,024 bytes). The JSON output limit applies to final JSON text, not peak allocation; Arrow IPC bodies use the separate binary-transfer limits above. For finite doubles, the cJSON writer first tries a 15-significant-digit representation and accepts it only when its own double read-back is exactly equal to the source value; otherwise it falls back to 17 significant digits. It releases each completed column tree while appending it to the output.

If a Paged response exceeds the output limit, the already-computed result is retried as progressively shorter row prefixes; the R query and user methods are not run again. If even one row cannot fit, `limit_exceeded` is returned with a warnings array, which may be empty.

Full sort and filter operations have no work or row limit by default because they are explicit user requests. An explicitly finite `jdv.max_work_bytes` or `jdv.max_query_rows` caps that work. Interrupt delivery in the event-loop path depends on the suspension state at callback entry: an interrupt caught during a started request returns `interrupted`, while R may defer delivery if the callback began with interrupts already suspended. The viewer UI should show that R is busy while a request runs.

Snapshot construction, Paged registration probes, page extraction, and JSON encoding use separate finite budgets. Each derived budget is the smaller of `INT_MAX`, six times the applicable output allowance plus 4 MiB, and an explicitly configured `jdv.max_work_bytes`. For Snapshot, the output allowance is `min(jdv.snapshot_max_bytes, jdv.max_message_bytes)`; for page work it is `jdv.max_message_bytes`. This preserves a 64 MiB work allowance with the 10 MiB defaults while allowing Snapshot construction to scale with its configured threshold. When `jdv.max_work_bytes` is finite, query work, page extraction, and response encoding draw from the same per-operation allowance. The cache budget limits the combined estimated bytes across column, channel, and index caches; oversized entries are skipped and old entries are evicted. The cache allowance is held by the registered table and is separate from per-operation work. Unfiltered page requests do not materialize a complete row index and are limited by page size.

These are conservative engine-work estimates, not process-memory or RSS limits: source storage, arbitrary user method allocations, R allocator and GC overhead, and transient R-internal string conversion or sorting scratch are outside the guarantee. Snapshot checks raw source strings before formatting and checks returned channel sizes before further encoding; Paged registration avoids scanning the full source and charges requested pages and queries. Oversized common metadata is rejected at registration, while explicit Paged row names are checked only for requested rows. If Auto selects Snapshot but full explicit row names exceed Snapshot's metadata budget, registration

retries as Paged; forced Snapshot registration returns `limit_exceeded`. Packed matrix columns are summarized by shape for display when copying a wide row would exceed the cell preview bound, and missingness scans are budgeted before allocating masks. Minimal control and error responses have a separate fixed 64 KiB encoder-work allowance, so a depleted request budget can still report its failure.

Registration API:

```
jdv_register(
  x,
  title = NULL,
  mode = c("auto", "snapshot", "paged"),
  collation = NULL
)

jdv_rpc(request = NULL)

jdv_view(x, title = NULL, mode = c("auto", "snapshot", "paged"),
  collation = NULL)

jdv_close(id = NULL)
```

`x` may be a data frame, matrix, or another R object. Data frames and matrices register as grid views. Arrow Table and RecordBatchReader objects, `nanoarrow_array_stream` pointers, and Polars DataFrame objects register as forward-only Arrow grid views through the optional `nanoarrow` producer adapter. Their mode and collation arguments do not change sequential access. Other objects use the tree view by default. `jdv_register()` returns `list(id, generation)` and does not open a socket. Its view/open message, and every other notification, is queued; see the Notification delivery section.

`jdv_view()` registers the view, opens the local connection on first use (unless `jdv_connect()` already did), sends hello on a new connection, then the queued notifications (view/closed, then view/open), and returns the view id invisibly. The R API keeps the return-list key `id` and generates opaque view id strings such as `v-1`; consumers must not interpret the id format. Its default title is the deparsed expression passed as `x`; an explicit `title = NULL` creates a new slot. `jdv_close(id)` closes one view, and `jdv_close()` closes every view. Closing the last view stops a connection that `jdv_view()` opened implicitly, after the close notification and any pending response are sent; a persistent connection from `jdv_connect()` stays open until `jdv_disconnect()`. `jdv_rpc()` with no argument or `NULL` explicitly polls the active connection and drains at most eight requests. This recovery path bypasses the normal event gate and is non-reentrant. Passing a JSON string to `jdv_rpc()` remains a synchronous, transport-independent embedding call.

The address is selected from `options(jdv.address = ...)`, then `JDV_ADDRESS`, then `discovery.json` under the user cache directory. On Linux and other Unix systems, the directory is `$XDG_CACHE_HOME/jdv` or `~/.cache/jdv`; on macOS it is `~/Library/Caches/jdv`; on Windows it is `%LOCALAPPDATA%/jdv` or `%USERPROFILE%/AppData/Local/jdv`. The discovery file is at most 64 KiB and contains a string field named `socketPath`. Unix-domain socket paths may be raw absolute paths or `unix:///absolute/path`; Windows named-pipe addresses use `npipe://./pipe/NAME`. A `tcp://localhost:port`, `tcp://127.0.0.1:port`, or `tcp://[:1]:port` address is accepted only when explicitly configured. TCP has no authentication, and named-pipe ACLs are not hardened. Windows named pipes must use byte-type mode; message-type pipes are unsupported.

Host support depends on the R front end's event loop and interrupt behavior. Rgui, RStudio, R.app, Esc/Ctrl-C delivery, and true top-level behavior must be validated in each target host. See [jdv_view\(\)](#) for the local transport execution gate.

A non-NULL title identifies one active display slot. Registering that active title again replaces its contents, reuses its view id, and increments its generation. The new view is prepared before the prior view is replaced, so a failed registration leaves the prior registration intact. Closing or evicting a title removes its title mapping; registering it later creates a new id at generation 1. NULL titles always create independent registrations. A new slot starts at generation 1. If `jdv.max_tables` is lowered below the active count, registration evicts as many oldest entries as needed to satisfy it.

Snapshot registration materializes and encodes its data in the open message. After successful message construction, a Snapshot with drillable columns retains its source and registration fingerprint in R for cell inspection. Other Snapshots retain only table metadata. The host is responsible for caching the snapshot data in both cases, and neither has a grid/rows endpoint. Paged registration retains the source and its initial schema in R and returns schema and dimensions in open; rows are computed on demand.

For Paged tables, dimensions, column names, and column classes are checked against the registration fingerprint on requests. Mutation of values through a shared reference cannot be detected reliably. Register the table again whenever values, schema, methods, or options that affect extracted channels change. Re-registration creates a fresh holder and cache. Changing a class method while continuing to use already cached channels does not guarantee refresh.

The default collation is `getOption("jdv.collation", "C")`. Current options include `jdv.max_message_bytes` (default 10 MiB, minimum 1,024), `jdv.max_request_bytes` (default 1 MiB, minimum 128), `jdv.max_tables` (default 64), `jdv.max_page_rows` (default 2,000), `jdv.snapshot_max_cells` (default 10,000), `jdv.snapshot_max_bytes` (default 10 MiB), `jdv.max_work_bytes` (default Inf), `jdv.max_cache_bytes` (default 32 MiB), and `jdv.max_query_rows` (default Inf). The work and row limits apply to full sort/filter operations; setting either option to a finite value restores the corresponding `limit_exceeded` guard. Page, work, cache, and query-row settings are captured when a Paged holder is registered; re-register the table to apply changed options. Each open Paged view separately retains its latest filtered/sorted row index outside `jdv.max_cache_bytes`, allowing consecutive pages for that query to reuse the index. A different query or unfiltered page releases the latest-index slot; an older result may remain in the LRU index cache. Clearing query caches, replacing the view generation, or closing the view releases both the slot and caches. This adds at most one potentially large result index per open Paged view; the 32 MiB default remains the budget for the LRU caches, including older result indices.

`mode = "auto"` selects Snapshot only when both the cell estimate $nrow * \max(1, ncol)$ and a deterministic byte estimate fit their thresholds. The byte threshold is $\min(jdv.snapshot_max_bytes, jdv.max_message_bytes)$. The cell estimate counts `source_row` even for a zero-column table. Byte estimation samples at most 256 evenly spaced rows; snapshot construction then checks actual source string lengths and generated channel sizes. A sampled estimate can therefore miss a large value, in which case automatic mode discards the incomplete Snapshot attempt and registers a Paged holder. Forced Snapshot mode returns `limit_exceeded` instead.

Binding tracking and `view/update_failed`:

Protocol contract. A view that follows a variable carries a tracking object in its `view/open` (see Message envelopes). While the state is "active", the view is rebuilt with reason: "tracked" and the next generation whenever the variable holds a different object than the view was built from, and that object becomes the new baseline. While the state is "paused", the variable is not examined, so

changes do not rebuild the view. When following ends, the state becomes "lost" with a reason, the last view stays, and view/tracking announces it; a later view/refresh fails with unavailable. The following sections describe how the R package implements this.

R implementation. A view can follow the variable it was created from. `jdv_view()` does so by default when its argument is a single name, and `jdv_register()` accepts an explicit `binding = list(name = "x", env = env)` for embedding hosts. After each top-level R task, the R task callback reads every tracked variable without evaluating it, compares the object it holds by identity with the one the view was built from, and rebuilds every changed view through the same update as view/refresh: a new view/open with the next generation (and `parent_closed` closes of its children) is queued, and the baseline moves to the new object. Nothing is scheduled beyond the task callback; a check is skipped while a request or a refresh is running and made at the next opportunity. When the built-in connection owns the session the queue is sent right after the check.

Only plain and already forced bindings are read (R 4.6.0 or later). Promise, active, and missing bindings are skipped and looked at again later. If the variable is removed or its environment is garbage collected, tracking ends, the last view stays, and a later view/refresh fails with unavailable; a recreated variable is not picked up. Objects changed in place (environments, reference objects, `data.table` updates by reference) keep their identity and are not detected; a manual view/refresh covers them. The environment is held weakly, but a displayed object that refers to it, such as a closure, keeps it alive. Views opened from another view and sequential views are not tracked. A paused view is skipped by the task callback, which exists only while at least one view is active.

Embedding hosts. R runs task callbacks in registration order. A host whose own callback takes notifications before the callback of jdv would see an update only after the next command. Such a host calls `jdv_rpc(NULL)` first, which checks tracked variables once even without a viewer connection, and then `jdv_notifications()`.

When an update cannot be built (a schema, conversion, or size limit, or an R error), the previous generation stays and one notification is queued:

```
{ "method": "view/update_failed",
  "params": { "view": "v-1", "generation": 3, "code": "limit_exceeded",
              "message": "..."} }
```

generation is the generation still displayed and code is one of the usual error codes. The same object is not tried again, and so not reported again, until the variable holds a different object or view/refresh is requested. A host that forwards the notification ignores one for a generation it no longer shows. A notification queue overflow ends the session as described under Notification delivery.

Registry-free R handles:

`jdv_handle()` captures a data frame, matrix, or list for an embedding host without registering a viewer slot. Handle operations do not consume `jdv.max_tables`, advance viewer identifiers, evict views, or produce notifications. `jdv_handle_register()` is the exception: it registers the captured value as an ordinary view, with the same registry limit, return value, notification, and transport flush behavior as `jdv_register()`. It uses the captured value directly and does not track a variable binding. The resulting view has a lifetime independent of the handle. Handles work without a session and alongside either an embedded session or a viewer connection. Closing registered views or disconnecting does not close handles. Sequential sources, including Arrow and Polars sources, are unsupported.

`jdv_handle_descriptor()` returns a JSON string with the shape of view/open params (or the object/open result for a child). `jdv_handle_request()` accepts only grid/rows and object/children, with the same parameter validation and result shapes as their viewer requests. It returns the result itself as JSON, without a response envelope. Omit view and generation from its R-list params: the handle supplies them, and specifying either is an `invalid_request` error. Result identifiers are opaque handle identifiers, separate from viewer identifiers, with generation 1. They cannot be used with `jdv_rpc()` or `jdv_close()`.

`jdv_handle_register()` uses the handle's mode and collation intent when those arguments are NULL (the default), and reuses its title unless a title is supplied. A closed handle or parent signals `view_not_found`, an operation on a busy handle signals `in_use`, and a changed table schema signals `schema_changed`.

`jdv_handle_open()` takes a typed path and returns a child R handle; call `jdv_handle_descriptor()` on that child to obtain its descriptor JSON. Other methods passed to `jdv_handle_request()`, including object/open, fail with `method_not_found`. These functions add no wire methods or fields; there is no view/describe method.

Use `mode = "paged"` for an initial preview bounded by rows, selected cols, and output bytes. The `limit` and `cols` params select the page; `max_bytes` bounds the UTF-8 JSON result. Oversized row or child pages shrink by halving their returned item count. A single item that cannot fit fails with `limit_exceeded`. Descriptors are never truncated. Auto mode retains the usual Snapshot selection, and Snapshot descriptors contain the complete captured data and do not support grid/rows.

Handles do not follow variable bindings or support refresh or tracking; their descriptor capabilities report both as false. Table handles retain the captured source, including in Snapshot mode, and check its dimensions, column names, and column classes against the initial fingerprint. This does not detect every value-only mutation through a shared reference or changes to methods. Close and recreate a handle when those change.

`jdv_handle_close()` closes the handle and all its descendants, is idempotent, and emits no notifications. Subsequent use fails with `view_not_found`. A live child retains its parent so that garbage collection of a parent variable does not invalidate the child. Explicit parent close still invalidates descendants. Unreachable handles release their resources through finalizers. Handles belong to the current R session and are not transferable between processes. See `jdv_handle()` for examples and limits.

Notification delivery:

Every notification R produces for a session (hello, view/open, view/closed, including evictions, cascaded child closes, the view/open of a child created by object/open, view/update_failed, and view/tracking) goes through one first-in, first-out queue. A session has exactly one owner of that queue, reported by the `owner` field of `jdv_connection()`:

- "transport": the built-in connection, whenever one is open. The transport sends the queue itself, before the response of the request being handled, so a notification always precedes the response that caused it.
- "embedded": a session started by `jdv_register()` when no connection exists. hello is queued before the first view/open. The host takes the queue with `jdv_notifications()`, which returns the JSON messages in order. The session remains after its last view closes so the final view/closed can be taken, and ends with `jdv_disconnect()`.

Ownership never moves implicitly: `jdv_connect()` and `jdv_view()` fail while an embedded session exists, and `jdv_notifications()` fails while the transport owns the queue. An embedding

host delivers every message taken by `jdv_notifications()` in order, and delivers the string returned by `jdv_rpc()` only after the notifications produced by that request. For object/open, the child's view/open is in the queue when `jdv_rpc()` returns, so the host registers the child from that notification and uses the response only to find it. `jdv_rpc()` never embeds notifications in its response.

```
id <- jdv_register(mtcars)$id
deliver(jdv_notifications())      # hello, view/open
response <- jdv_rpc(request_json) # a request received from the viewer
deliver(jdv_notifications())      # notifications caused by the request
deliver(response)
```

The queue is bounded by `jdv.notification_max_messages` (default `max(256, 4 * jdv.max_tables)`) and `jdv.notification_max_bytes` (default `4 * jdv.max_message_bytes`), both read when the session starts. Messages are never dropped silently: when adding one would exceed a limit, the session ends, every view is closed, the queue is discarded, and `last_disconnect_reason` of `jdv_connection()` becomes "notification_overflow". The operation that overflowed fails with an error of class `jdv_notification_overflow`. With the built-in transport the queue is normally sent right away, so an overflow mostly affects an embedded host that does not take its notifications. A host must treat the end of a session, seen as an error from R or as `last_disconnect_reason`, as the loss of every view.

Local transport and execution gate:

The following limits and dispatch rules describe the R producer transport; another producer may implement its execution gate differently. The R transport bounds the pending queue to 256 messages and 16 MiB by default. Each inbound JSON line is limited by `jdv.max_request_bytes` (1 MiB by default), and each outbound line by `jdv.max_message_bytes` (10 MiB by default). A read callback processes at most `max(64 KiB, request_limit + 1)` bytes. One native poll or event callback drains at most eight queued requests; each request in that batch passes through its own execution-gate check. Sends have a one-second timeout. Limits can be adjusted with `jdv.transport.queue_messages`, `jdv.transport.queue_bytes`, `jdv.transport.max_drain_requests`, and `jdv.transport.send_timeout_ms`.

The event source reads and validates framed input into a bounded C queue. It does not leave a readable descriptor active after hitting its receive quota. If the pending queue is full by message count or bytes, the transport returns `limit_exceeded` for that request alone and keeps the connection and already queued work when the error response is sent successfully. A failed error send still disconnects the host and closes registered tables. Each receive callback is also bounded to `jdv.transport.queue_messages + jdv.transport.max_drain_requests` frames and `max(64 KiB, request_limit + 1)` bytes. cancel and wake notifications count toward the frame limit. A host that exceeds either callback quota is disconnected. Each receive callback acknowledges the cumulative number of complete frames read, including malformed or rejected frames; the host releases frame and byte credit through that boundary. Since the R producer sets `hello.params.wake` to true, hosts must send wake notifications repeatedly at a regular interval while a request awaits a terminal response. A single wake is insufficient for this R execution gate: a normal package API-entry preflight is receive-only and does not dispatch work, and receiving a wake while R is in a function frame only allows requests to be queued for later dispatch. A 50 ms interval is a valid implementation choice, but the protocol does not require every host to use that interval. Wake frames remain subject to the same applicable credit and rate limits as other traffic. Hosts must keep at most 32 view requests awaiting a response. Before sending a page request that makes an unsent page request obsolete, hosts must

replace the unsent request with the newer page. Hosts must bound the cumulative unconfirmed output prefix to the receive callback's frame and byte quotas so multiple writes cannot exceed one callback's budget when coalesced. Each cancel and wake frame consumes the same frame and byte credit as a request. Before dispatch, the transport removes cancelled and expired queued requests and sends their terminal errors, including while another R method is running. The R adapter returns `view_not_found` for a closed or missing view and `stale_generation` for an obsolete generation. For each request, dispatch requires `native_busy == 0` and a single `base::sys.nframe()` evaluation in `R_BaseEnv` to return zero. If the queue is empty, the gate skips that evaluation. A started request that fails, including an interrupt, is terminal and is never retried. A synchronous or queued stale generation returns `stale_generation`. A second live pending request with the same typed id is rejected as `invalid_request` on a best-effort basis and closes the connection to avoid ambiguous responses. Cancelled requests no longer reserve their ids.

The gate temporarily saves `R_interrupts_suspended`, sets it to `TRUE` for the single frame-count evaluation, and restores the prior value with unwind-safe cleanup. It does not use `END_SUSPEND_INTERRUPTS`, call `onintr`, or inspect or change private pending-interrupt state. This practical function-frame check is not a guarantee that R is idle: on Windows, a timer can run during a top-level for loop while `base::sys.nframe()` is zero, and this dispatch behavior has been observed. The Windows timer interval is about 200 ms. If the request callback begins with interrupts enabled, an interrupt after dequeue becomes an interrupted error. If R entered the callback with interrupts already suspended, R may defer delivery; long sort or filter work may therefore not stop promptly. The viewer should show a busy state and offer a retry after a terminal failure.

`jdv_rpc(NULL)` is a non-reentrant recovery path that bypasses the normal gate and drains at most the configured batch under `R_UnwindProtect`; it does not use a background `R_TopLevelExec`. Disconnect cleanup is deferred until the callback stack is safe. A public R `base::addTaskCallback()` performs cleanup after a successful top-level task, while explicit polling and later public package entries also check for pending cleanup. This task callback is cleanup-only and does not dispatch requests. It is an event-driven cleanup opportunity, not a general timer. A disconnect closes every registered table, and the next `jdv_view()` call starts a new connection; no in-flight request is retried.

On namespace unload, the transport source is stopped, queued requests and registry entries are closed, and the housekeeping callback is removed. If a native callback is still active, the package namespace may unload while the native library remains loaded while native calls are active; an idle reload or unload can release it later. Forced DLL unload while native callbacks are active is unsupported; reloading is rejected while the previous native owner is still active.

R evaluation, condition handling, and interrupt state use installed public R APIs. The POSIX input handler uses the installed `R_ext/eventloop.h` external interface, which is not a stable API. No GPL R source or later implementation is copied.

See Also

[jdv_register\(\)](#), [jdv_rpc\(\)](#), [jdv_view\(\)](#), [jdv_close\(\)](#), [jdv_query_spec](#)

jdv_view

Open an object in the local jdv viewer

Description

Open an object in the local jdv viewer

Usage

```
jdv_view(
  x,
  title = NULL,
  mode = c("auto", "snapshot", "paged"),
  collation = NULL,
  track = TRUE
)
```

Arguments

<code>x</code>	A data frame or matrix for grid views; an Arrow Table or RecordBatchReader, nanoarrow array stream, or Polars DataFrame for a sequential grid; or an R object to explore in a tree view.
<code>title</code>	Display title. When omitted, the expression passed as <code>x</code> is used; explicitly passing <code>NULL</code> creates an independent table slot.
<code>mode</code>	Grid view mode: "auto", "snapshot", or "paged".
<code>collation</code>	Grid string ordering policy, "C" or "locale". When <code>NULL</code> (the default), <code>getOption("jdv.collation", "C")</code> supplies the default. A non- <code>NULL</code> value is kept as the view's refresh intent.
<code>track</code>	Whether to follow the variable passed as <code>x</code> ; see Details. To follow a variable on behalf of a caller, see jdv_binding() .

Details

In a grid view, every column is displayed, but only some columns can be filtered by value (for example logical, numeric, character, factor, date, date-time, duration, and integer64 columns). Others, such as list columns and columns of unsupported classes, can only be tested for missing values. See the R column mapping section of [jdv_query_spec](#) for which columns are supported by the current release.

The viewer connection is opened on the first call unless [jdv_connect\(\)](#) already opened one; in that case the existing connection is used and hello is not sent again. An address can be set in `options(jdv.address = ...)` or `JDV_ADDRESS`; otherwise `jdv` reads `discovery.json` from the user cache directory. The discovery file is limited to 64 KiB and contains a `socketPath` string. Local Unix-domain sockets and Windows named pipes are used by default; TCP is available only when an explicit address selects it. A connection opened implicitly by this function is closed after the last table is closed; a persistent connection from [jdv_connect\(\)](#) stays open until [jdv_disconnect\(\)](#).

The event gate dispatches only when `sys.nframe() == 0`. This practical function-frame check is not a guarantee that R is idle: on Windows, a timer can run during top-level computation while the function-frame count is zero. A request that reaches an interrupt while running fails with interrupted and is never retried automatically. While a sort or filter runs, the event gate restores the interrupt-suspension state it found before calling R methods. If interrupts were already suspended on entry, R may defer Ctrl-C during a long sort or filter until a later check; use the viewer's busy state as an indication that the request is still running. A bare top-level [base::browser\(\)](#) can service requests; a frontend waiting inside a function frame, such as a function that calls [base::browser\(\)](#), must let that frame unwind or explicitly call `jdv_rpc(NULL)` before queued work can progress. A script that waits in a loop never returns to top level, so it must poll with

`jdv_rpc(NULL)`; see `jdv_rpc()`. Windows GUI and Esc behavior have not been validated. Windows named pipes must use byte-type mode; message-type pipes are unsupported. The named-pipe transport, including binary Arrow pages and requests serviced while an interactive session is idle, is tested on Windows through Rscript and the arf console; Rgui and Rterm are not tested. Named-pipe ACLs are not hardened, and an explicitly selected TCP endpoint has no authentication. Registry entries created with `jdv_register()` share this process-wide registry and are also released if the viewer disconnects. Package unload stops the event source and pending work before releasing table references.

Tracking. With `track = TRUE` (the default), a call whose argument is a single variable name, such as `jdv_view(df)`, follows that variable. After each top-level R task jdv looks at the variable, and when it holds a different object (for example after `df <- transform(df, ...)`) every changed view is rebuilt as its next generation and the viewer updates by itself. Expressions such as `jdv_view(head(df))` are never re-evaluated and are not tracked; pass `track = FALSE` for a fixed view. The variable is looked up once, without evaluating anything, in the calling environment and then outward, and the view stays bound to the environment where the name was found: a later variable of the same name in an inner scope is ignored, and inside `f <- function(x) jdv_view(x)` the argument `x` of `f` is followed.

Detection compares object identity. An object that is modified in place, such as an environment, a reference class or R6 object, or a `data.table` changed with `:=`, keeps its identity and is not noticed; use the viewer's refresh. Assigning the same object again does nothing. Promise and active bindings are never evaluated by the check: they are skipped and looked at again after later tasks. When the variable is removed or its environment is garbage collected, tracking ends and the last view stays; the view can then no longer be refreshed, and re-running `jdv_view()` tracks again. The environment is only weakly referenced, but a displayed object that refers to it, for example a closure, keeps it alive. Views opened from a view, and sequential sources, are not tracked. If an update fails (for example the new object exceeds a limit) the previous view stays and one `view/update_failed` notification is sent; the same object is not retried until the variable changes or the view is refreshed manually. The viewer host can stop and resume following a view with the `view/track` request (see `jdv_spec`).

Tracking needs R 4.6.0 or later. Where it is not possible, including when the variable cannot be read safely, the view is still shown and a warning says it is not followed. A manual `view/refresh` request rebuilds from the current value of a tracked variable. Tracking is carried out by an R task callback that exists only while a tracked view does and is independent of the viewer connection. An embedding host that takes notifications with `jdv_notifications()` should call `jdv_rpc(NULL)` and then `jdv_notifications()` from its own task callback, because a callback that runs before the one of jdv would otherwise see updates only after the next command; see `jdv_rpc()`.

Value

The view id, invisibly.

Index

base::addTaskCallback(), 45
base::browser(), 46
base::class(), 17, 38
base::difftime(), 17
base::format(), 17, 18
base::I(), 18
base::order(), 18
base::sys.nframe(), 45
base::Sys.sleep(), 21
base::typeof(), 38
base::unclass(), 14
base::xtfrm(), 17, 18

jdv_binding, 2
jdv_binding(), 2, 5, 20, 46
jdv_close, 3
jdv_close(), 5, 36, 38, 40, 43, 45
jdv_connect, 4
jdv_connect(), 3–5, 40, 43, 46
jdv_connection (jdv_connect), 4
jdv_connection(), 4, 9, 43, 44
jdv_disconnect (jdv_connect), 4
jdv_disconnect(), 4, 5, 9, 36, 40, 43, 46
jdv_errors, 2, 4, 5, 8, 9, 14
jdv_handle, 6
jdv_handle(), 6–8, 42, 43
jdv_handle_close (jdv_handle), 6
jdv_handle_close(), 8, 43
jdv_handle_descriptor (jdv_handle), 6
jdv_handle_descriptor(), 8, 43
jdv_handle_open (jdv_handle), 6
jdv_handle_open(), 6–8, 43
jdv_handle_register (jdv_handle), 6
jdv_handle_register(), 7, 8, 42, 43
jdv_handle_request (jdv_handle), 6
jdv_handle_request(), 6, 8, 43
jdv_notifications, 9
jdv_notifications(), 3, 5, 9, 21, 42–44, 47
jdv_protocol (jdv_spec), 22
jdv_query_spec, 6, 10, 45, 46

jdv_register, 19
jdv_register(), 2–5, 7–9, 34, 36, 40, 42, 43, 45, 47
jdv_rpc, 21
jdv_rpc(), 5, 9, 20, 38–40, 43–45, 47
jdv_spec, 5, 9, 17, 19, 21, 22, 47
jdv_view, 45
jdv_view(), 2–5, 20, 21, 40–43, 45, 47

query specification, 24, 28, 30, 32, 35