Package 'orderly'

October 9, 2025

Title Lightweight Reproducible Reporting

Version 2.0.0

Description Distributed reproducible computing framework, adopting ideas from git, docker and other software. By defining a lightweight interface around the inputs and outputs of an analysis, a lot of the repetitive work for reproducible research can be automated. We define a simple format for organising and describing work that facilitates collaborative reproducible research and acknowledges that all analyses are run multiple times over their lifespans.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

URL https://github.com/mrc-ide/orderly

BugReports https://github.com/mrc-ide/orderly/issues

Imports R6, cli, diffobj, fs, gert (>= 1.9.3), httr2 (>= 1.2.1), jsonlite, openssl, rlang, rstudioapi, vctrs, withr, yaml

Suggests DBI, RSQLite, callr, jsonvalidate (>= 1.4.0), knitr, mockery, pkgload, processx, rmarkdown, testthat (>= 3.0.0), webfakes

Config/testthat/edition 3

VignetteBuilder knitr

Language en-GB

NeedsCompilation no

Author Rich FitzJohn [aut, cre],
Robert Ashton [aut],
Martin Eden [aut],
Alex Hill [aut],
Wes Hinsley [aut],
Mantra Kusumgar [aut],
Paul Liétar [aut],
James Thompson [aut],

Katy Gaythorpe [aut],
Imperial College of Science, Technology and Medicine [cph]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2025-10-09 08:40:09 UTC

Contents

`orderly_artefact` *Declare orderly artefacts*

Description

Declare an artefact. By doing this you turn on a number of orderly features; see Details below. You can have multiple calls to this function within your orderly script.

Usage

```
orderly_artefact(description = NULL, files)
```

Arguments

`description`	The name of the artefact
`files`	The files within this artefact

Details

(1) files matching this will *not* be copied over from the src directory to the draft directory unless they are also listed as a resource with `orderly_resource()`. This feature is only enabled if you call this function from the top level of the orderly script and if it contains only string literals (no variables).

(2) if your script fails to produce these files, then `orderly_run()` will fail, guaranteeing that your task does really produce the things you need it to.

(3) within the final metadata, your artefacts will have additional metadata; the description that you provide and a grouping

Value

Undefined

Examples

```
# An example in context within the orderly examples:
orderly_example_show("strict")
```

`orderly_cleanup` *Clean up source directory*

Description

Find, and delete, file that were generated by running a report. Until you're comfortable with what this will do, you are strongly recommended to run `orderly_cleanup_status` first to see what will be deleted.

Usage

```
orderly_cleanup(name = NULL, dry_run = FALSE, root = NULL)

orderly_cleanup_status(name = NULL, root = NULL)
```

Arguments

`name`	Name of the report directory to clean (i.e., we look at `src/<name>` relative to your orderly root
`dry_run`	Logical, indicating if we should *not* delete anything, but instead just print information about what we would do
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function **does** require that the directory is configured for orderly, and not just outpack (see `orderly_init()` for details).

Details

After file deletion, we look through and remove all empty directories; orderly has similar semantics here to git where directories are never directly tracked.

For recent `gert` we will ask git if files are ignored; if ignored then they are good candidates for deletion! We encourage you to keep a per-report `.gitignore` that lists files that will copy into the source directory, and then we can use that same information to clean up these files after generation. Importantly, even if a file matches an ignore rule but has been committed to your repository, it will no longer match the ignore rule.

Value

An (currently unstable) object of class `orderly_cleanup_status` within which the element `delete` indicates files that would be deleted (for `orderly_cleanup_status`) or that were deleted (for `orderly_cleanup`)

Notes for user of orderly1

In orderly1 this function has quite different semantics, because the full set of possible files is always knowable from the yaml file. So there, we start from the point of view of the list of files then compare that with the directory.

Examples

```
# Create a simple example:
path <- orderly_example()

# We simulate running a packet interactively by using 'source';
# you might have run this line-by-line, or with the "Source"
# button in Rstudio.
source(file.path(path, "src/data/data.R"), chdir = TRUE)

# Having run this, the output of the report is present in the
# source directory:
fs::dir_tree(path)

# We can detect what might want cleaning up by running
# "orderly_cleanup_status":
orderly_cleanup_status("data", root = path)

# Soon this will print more nicely to the screen, but for now you
# can see that the status of "data.rds" is "derived", which means
# that orderly knows that it is subject to being cleaned up; the
# "delete" element shows what will be deleted.

# Do the actual deletion:
orderly_cleanup("data", root = path)
```

---

orderly_compare_packets

Compare the metadata and contents of two packets.

Description

Insignificant differences in the metadata (e.g., different dates and packet IDs) are excluded from the comparison.

Usage

```
orderly_compare_packets(
  target,
  current,
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE,
  root = NULL
)
```

Arguments

target	The id of the packet to use in the comparison.

current	The id of the other packet against which to compare.
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

An object of class orderly_comparison. The object can be printed to get a summary description of the differences, or passed to orderly_comparison_explain() to display more details.

Examples

```
# Here are two packets that are equivalent, differing only in id
# and times:
path <- orderly_example()
id1 <- orderly_run("data", root = path)
id2 <- orderly_run("data", root = path)
orderly_compare_packets(id1, id2, root = path)

# A more interesting comparison:
id1 <- orderly_run("parameters", list(max_cyl = 6), root = path)
id2 <- orderly_run("parameters", list(max_cyl = 4), root = path)
cmp <- orderly_compare_packets(id1, id2, root = path)
cmp

# A verbose comparison will show differences in the constituent
# components of each packet:
orderly_comparison_explain(cmp, verbose = TRUE)
```

---

orderly_comparison_explain

Print the details of a packet comparison.

Description

This function allows to select what part of the packet to compare, and in how much details.

Usage

```
orderly_comparison_explain(cmp, attributes = NULL, verbose = FALSE)
```

Arguments

cmp	An orderly_comparison object, as returned by orderly_compare_packets().
attributes	A character vector of attributes to include in the comparison. The values are keys of the packets' metadata, such as parameters or files. If NULL, the default, all attributes are compared, except those that differ in trivial way (i.e., id and time).
verbose	Control over how much information is printed. It can either be a logical, or a character scalar silent or summary.

Value

Invisibly, a logical indicating whether the packets are equivalent, up to the given attributes.

Examples

```
path <- orderly_example()
id1 <- orderly_run("parameters", list(max_cyl = 6), root = path)
id2 <- orderly_run("parameters", list(max_cyl = 4), root = path)
cmp <- orderly_compare_packets(id1, id2, root = path)

orderly_comparison_explain(cmp)
orderly_comparison_explain(cmp, verbose = TRUE)
orderly_comparison_explain(cmp, "parameters", verbose = TRUE)
```

---

orderly_config *Read configuration*

Description

Read the current orderly configuration, stored within the outpack root, along with any orderly-specific extensions.

Usage

```
orderly_config(root = NULL)
```

Arguments

root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

A list of configuration options:

- core: The most important options about the outpack store, containing:
  - path_archive: The path to the human-readable packet archive, or NULL if disabled (set in orderly_config_set() as core.path_archive)
  - use_file_store: Indicates if a content-addressable file store is enabled (core.use_file_store)
  - require_complete_tree: Indicates if this outpack store requires all dependencies to be fully available (core.require_complete_tree)
  - hash_algorithm: The hash algorithm used (currently not modifiable)
- location: Information about locations; see orderly_location_add(), orderly_location_rename() and orderly_location_remove() to interact with this configuration, or orderly_location_list() to more simply list available locations. Returns as a data.frame with columns name, id, priority, type and args, with args being a list column.
- orderly: A list of orderly-specific configuration; this is just the minimum required version (as minimum_orderly_version).

Examples

```
# A default configuration in a new temporary directory
path <- withr::local_tempdir()
orderly_init(path)
orderly_config(path)
```

---

orderly_config_set *Set configuration options*

Description

Set configuration options. Not all can currently be set; this will be expanded over time. See Details.

Usage

```
orderly_config_set(..., options = list(...), root = NULL)
```

Arguments

...	Named options to set (e.g., pass the argument `core.require_complete_tree = TRUE`)
options	As an alternative to ..., you can pass a list of named options here (e.g., `list(core.require_complete_t = TRUE)`). This interface is typically easier to program against.
root	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Details

Options are set in the order that they are provided. Currently, if setting one option fails, no further options will be processed but previous ones will be (do not rely on this behaviour, it may change).

Currently you can set:

- `core.require_complete_tree`

See `orderly_init()` for description of these options.

Value

Nothing

See Also

orderly_config

Examples

```
# The default configuration does not include a file store, and
# saves output within the "archive" directory:
path <- withr::local_tempdir()
orderly_init(path)
fs::dir_tree(path, all = TRUE)

# Change this after the fact:
orderly_config_set(core.use_file_store = TRUE,
                   core.path_archive = NULL,
                   root = path)
fs::dir_tree(path, all = TRUE)
```

`orderly_copy_files` *Copy files from a packet*

Description

Copy files from a packet to anywhere. Similar to `orderly_dependency()` except that this is not used in an active packet context. You can use this function to pull files from an outpack root to a directory outside of the control of outpack, for example. Note that all arguments need must be provided by name, not position, with the exception of the id or query.

Usage

```
orderly_copy_files(
  expr,
  files,
  dest,
  overwrite = TRUE,
  name = NULL,
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE,
  parameters = NULL,
  options = NULL,
  envir = parent.frame(),
  root = NULL
)
```

Arguments

`expr`	The query expression. A `NULL` expression matches everything.
`files`	Files to copy from the other packet, as a character vector. If the character vector is unnamed, the files listed are copied over without changing their names. If the vector is named however, the names will be used as the destination name for the files. In either case, if you want to import a directory of files from a packet, you must refer to the source with a trailing slash (e.g., `c(here = "there/")`), which will create the local directory `here/...` with files from the upstream packet directory `there/`. If you omit the slash then an error will be thrown suggesting that you add a slash if this is what you intended. You can use a limited form of string interpolation in the names of this argument; using `${variable}` will pick up values from `envir` and substitute them into your string. This is similar to the interpolation you might be familiar with from `glue::glue` or similar, but much simpler with no concatenation or other fancy features supported. Note that there is an unfortunate, but (to us) avoidable inconsistency here; interpolation of values from your environment in the query is done by using `environment:x` and in the destination filename by doing `${x}`.

	If you want to copy *all* files from the packet, use `./` (read this as the directory of the packet). The trailing slash is required in order to be consistent with the rules above.
`dest`	The directory to copy into
`overwrite`	Overwrite files at the destination; this is typically what you want, but set to `FALSE` if you would prefer that an error be thrown if the destination file already exists.
`name`	Optionally, the name of the packet to scope the query on. This will be intersected with `scope` arg and is a shorthand way of running `scope = list(name = "name")`
`location`	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
`allow_remote`	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is `TRUE`, then in conjunction with `orderly_dependency()` you might pull a large quantity of data. The default is `NULL`. This is `TRUE` if remote locations are listed explicitly as a character vector in the `location` argument, or if you have specified `fetch_metadata = TRUE`, otherwise `FALSE`.
`fetch_metadata`	Logical, indicating if we should pull metadata immediately before the search. If `location` is given, then we will pass this through to `orderly_location_fetch_metadata()` to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to `FALSE` after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
`parameters`	Optionally, a named list of parameters to substitute into the query (using the `this:` prefix)
`options`	**DEPRECATED**. Please don't use this any more, and instead use the arguments `location`, `allow_remote` and `fetch_metadata` directly.
`envir`	Optionally, an environment to substitute into the query (using the `environment:` prefix). The default here is to use the calling environment, but you can explicitly pass this in if you want to control where this lookup happens.
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Details

You can call this function with an id as a string, in which case we do not search for the packet and proceed regardless of whether or not this id is present. If called with any other arguments (e.g., a string that does not match the id format, or a named argument `name`, `subquery` or `parameters`) then we interpret the arguments as a query and `orderly_search()` to find the id. It is an error if this query does not return exactly one packet id, so you probably want to use `latest()`.

There are different ways that this might fail (or recover from failure):

- if `id` is not known in the metadata store (not known because it's not unpacked but also not known to be present in some other remote) then this will fail because it's impossible to resolve

the files. Consider refreshing the metadata with `orderly_location_fetch_metadata()` to refresh this.

- if the `id` is not unpacked *and* no local copy of the files referred to can be found, we error by default (but see the next option). However, sometimes the file you refer to might also be present because you have downloaded a packet that depended on it, or because the content of the file is unchanged because from some other packet version you have locally.
- if the `id` is not unpacked, there is no local copy of the file and if `allow_remote` is `TRUE` we will try and request the file from whatever remote would be selected by `orderly_location_pull()` for this packet.

Note that empty directories might be created on failure.

Value

Primarily called for its side effect of copying files from a packet into the directory `dest`. Also returns a list with information about the copy, containing elements:

- `id`: The resolved id of the packet
- `name`: The name of the packet
- `files`: a data.frame of filenames with columns `here` (the name of the file in `dest`) and `there` (the name of the file in the packet)

Examples

```
root <- orderly_example()
orderly_run("data", root = root)

dest <- withr::local_tempdir()
res <- orderly_copy_files("latest", name = "data", "data.rds",
                          dest = dest, root = root)

# We now have our data in the destination directory:
fs::dir_tree(dest)

# Information about the copy:
res
```

`orderly_dependency` *Declare a dependency*

Description

Declare a dependency on another packet

Usage

```
orderly_dependency(name, query, files)
```

Arguments

`name`	The name of the packet to depend on
`query`	The query to search for; often this will simply be the string `latest`, indicating the most recent version. You may want a more complex query here though.
`files`	Files to copy from the other packet, as a character vector. If the character vector is unnamed, the files listed are copied over without changing their names. If the vector is named however, the names will be used as the destination name for the files. In either case, if you want to import a directory of files from a packet, you must refer to the source with a trailing slash (e.g., `c(here = "there/")`), which will create the local directory `here/...` with files from the upstream packet directory `there/`. If you omit the slash then an error will be thrown suggesting that you add a slash if this is what you intended. You can use a limited form of string interpolation in the names of this argument; using `${variable}` will pick up values from `envir` and substitute them into your string. This is similar to the interpolation you might be familiar with from `glue::glue` or similar, but much simpler with no concatenation or other fancy features supported. Note that there is an unfortunate, but (to us) avoidable inconsistency here; interpolation of values from your environment in the query is done by using `environment:x` and in the destination filename by doing `${x}`. If you want to copy *all* files from the packet, use `./` (read this as the directory of the packet). The trailing slash is required in order to be consistent with the rules above.

Details

See `orderly_run()` for some details about how search options are used to select which locations packets are found from, and if any data is fetched over the network. If you are running interactively, this will obviously not work, so you should use `orderly_interactive_set_search_options()` to set the options that this function will respond to.

Value

Undefined

Examples

```
orderly_example_show("depends")
```

`orderly_description` *Describe the current packet*

Description

Describe the current packet

Usage

```
orderly_description(display = NULL, long = NULL, custom = NULL)
```

Arguments

display	A friendly name for the report; this will be displayed in some locations of the web interface, packit. If given, it must be a scalar character.
long	A longer description of the report. If given, it must be a scalar character.
custom	Any additional metadata. If given, it must be a named list, with all elements being scalar atomics (character, number, logical).

Value

Undefined

Examples

```
# An example in context within the orderly examples:
orderly_example_show("data")
```

---

orderly_example *Copy a simple orderly example*

Description

Copy a simple orderly example for use in the docs. This function should not form part of your workflow!

Usage

```
orderly_example(..., names = NULL, example = "demo", dest = NULL)
```

Arguments

...	Arguments passed through to orderly_init()
names	Optionally, names of the reports to copy. The default is to copy all reports.
example	The name of the example to copy. Currently only "simple" and "demo" are supported.
dest	The destination. By default we use withr::local_tempdir() which will create a temporary directory that will clean itself up. This is suitable for use from the orderly examples, but you may prefer to provide your own path, in which case the path must not already exist.

Value

Invisibly, the path to the example.

Examples

```
path <- orderly_example()
orderly_list_src(root = path)
```

`orderly_example_show` *Show an example file*

Description

Show a file from within one of the examples. This function exists for use within orderly help files, vignettes and tutorials and is not meant to form part of your workflows, unless you are doing something very peculiar.

Usage

```
orderly_example_show(name, file = NULL, example = "demo")
```

Arguments

`name`	The name of the report within the example.
`file`	The name of the file within the report. The default is to show the main orderly file (i.e., `<name>.R`)
`example`	The name of the example to look in. The default `demo` is a sprawling set of source designed to show off different orderly features.

Details

All orderly examples here are runnable, though some will naturally have some pre-requisites (e.g., using a dependency will require that the dependency has been run first).

Value

Nothing, called for its side effects only.

Examples

```
# We use constructions like this in the help, to show off features
# of orderly:
orderly_example_show("data")

# You can run this example:
path <- orderly_example()
orderly_run("data", root = path)
```

`orderly_gitignore_update` *Update a gitignore file*

Description

Update a gitignore, which is useful to prevent accidentally committing files to source control that are generated. This includes artefacts, shared resources and dependencies (within a report directory) or at the global level all the contents of the `.outpack` directory, the draft folder and the archive directory.

Usage

```
orderly_gitignore_update(name, root = NULL)
```

Arguments

`name`	The name of the gitignore file to update, or the string "(root)"
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function **does** require that the directory is configured for orderly, and not just outpack (see `orderly_init()` for details).

Details

If this function fails with a message `Can't edit '.gitignore', markers are corrupted`, then look for the special markers within the `.gitignore` file. It should look like

```
# ---VVV--- added by orderly ---VVV----------------
# Don't manually edit content between these markers
... patterns
# ---^^^--- added by orderly ---^^^----------------
```

We can't edit the file if:

- any of these lines appears more than once in the file
- there is anything between the first two lines
- they are not in this order

If you get the error message, search and remove these lines and rerun.

Value

Nothing, called for its side effects

Examples

```
path <- orderly_example()

# Update core orderly ignorables:
orderly_gitignore_update("(root)", root = path)
cli::cli_code(readLines(file.path(path, ".gitignore")))

# Report-specific ignores:
orderly_gitignore_update("data", root = path)
cli::cli_code(readLines(file.path(path, "src", "data", ".gitignore")))
```

---

orderly_hash_file *Compute a hash*

Description

Use orderly's hashing functions. This is intended for advanced users, in particular those who want to create hashes that are consistent with orderly from within plugins. The default behaviour is to use the same algorithm as used in the orderly root (via the `root` argument, and the usual root location approach). However, if a string is provided for `algorithm` you can use an alternative algorithm.

Usage

```
orderly_hash_file(path, algorithm = NULL, root = NULL)

orderly_hash_data(data, algorithm = NULL, root = NULL)
```

Arguments

`path`	The name of the file to hash
`algorithm`	The name of the algorithm to use, overriding that in the orderly root.
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).
`data`	A string to hash

Value

A string in the format `<algorithm>:<digest>`

Examples

```
orderly_hash_data("hello", "md5")

# If you run this function from within the working directory of an
# orderly root, then you can omit the algorithm and it will use
# the algorithm used by orderly (which will be sha256):
path <- orderly_example()
withr::with_dir(path, orderly_hash_data("hello"))
```

`orderly_init` *Initialise an orderly repository*

Description

Initialise an empty orderly repository, or initialise a source copy of an orderly repository (see Details). An orderly repository is defined by the presence of a file `orderly_config.json` (or `orderly_config.yml`) at its root, along with a directory `.outpack/` at the same level.

Usage

```
orderly_init(
  root = ".",
  path_archive = "archive",
  use_file_store = FALSE,
  require_complete_tree = FALSE,
  force = FALSE
)
```

Arguments

`root`	The path to initialise the repository root at. If the repository is already initialised, this operation checks that the options passed in are the same as those set in the repository (erroring if not), but otherwise does nothing. The default path is the current working directory.
`path_archive`	Path to the archive directory, used to store human-readable copies of packets. If `NULL`, no such copy is made, and `file_store` must be `TRUE`
`use_file_store`	Logical, indicating if we should use a content-addressable file-store as the source of truth for packets. If `archive` is non-`NULL`, the file-store will be used as the source of truth and the duplicated files in archive exist only for convenience.
`require_complete_tree`	Logical, indicating if we require a complete tree of packets. This currently affects `orderly_location_pull()`, by requiring that it always operates in recursive mode. This is `FALSE` by default, but set to `TRUE` if you want your archive to behave well as a location; if `TRUE` you will always have all the packets that you hold metadata about.
`force`	Logical, indicating if we should initialise orderly even if the directory is not empty.

Details

It is expected that `orderly_config.json` will be saved in version control, but that `.outpack` will be excluded from version control; this means that for every clone of your project you will need to call `orderly_init()` to initialise the `.outpack` directory. If you forget to do this, an error will be thrown reminding you of what you need to do.

You can safely call `orderly_init()` on an already-initialised directory, however, any arguments passed through must exactly match the configuration of the current root, otherwise an error will be thrown. Please use `orderly_config_set()` to change the configuration within `.outpack`, as this ensures that the change in configuration is possible. If configuration options are given but match those that the directory already uses, then nothing happens. You can safely edit `orderly_config.json` yourself, at least for now.

If the repository that you call `orderly_init()` on is already initialised with an `.outpack` directory but not an `orderly_config.json` file, then we will write that file too.

Value

The full, normalised, path to the root, invisibly. Typically this is called only for its side effect.

Examples

```
# We'll use an automatically cleaned-up directory for the root:
path <- withr::local_tempdir()

# Initialise a new repository, setting an option:
orderly_init(path, use_file_store = TRUE)

fs::dir_tree(path, all = TRUE)
```

---

orderly_interactive_set_search_options

Set search options for interactive use

Description

Set search options for interactive use of orderly; see `orderly_dependency()` and `orderly_run()` for details. This applies only for the current session, but applies to all interactive uses of orderly functions that might have received a copy of the search options (`location`, `allow_remote` and `fetch_metadata`) via `orderly_run()`. Calling with no arguments resets to the defaults.

Usage

```
orderly_interactive_set_search_options(
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE
)
```

Arguments

location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.

Value

Nothing, called for its side effects

Examples

```
# enable fetching packets from remote locations in this session
orderly_interactive_set_search_options(allow_remote = TRUE)
# ... your interactive session
# reset to defaults
orderly_interactive_set_search_options()
```

---

orderly_list_src *List source reports*

Description

List source reports - that is, directories within src/ that look suitable for running with orderly; these will be directories that contain an entrypoint file - a .R file with the same name as the directory (e.g., src/data/data.R corresponds to data).

Usage

```
orderly_list_src(root = NULL)
```

Arguments

root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function **does** require that the directory is configured for orderly, and not just outpack (see orderly_init() for details).

Value

A character vector of names of source reports, suitable for passing to `orderly_run()`

See Also

`orderly_metadata_extract()` for listing packets that have completed

Examples

```
path <- orderly_example()
orderly_list_src(root = path)
```

`orderly_location_add` *Add a new location*

Description

Add a new location - a place where other packets might be found and pulled into your local archive. Currently only file and http based locations are supported, with limited support for custom locations. Note that adding a location does *not* pull metadata from it, you need to call `orderly_location_fetch_metadata()` first. The function `orderly_location_add` can add any sort of location, but the other functions documented here (`orderly_location_add_path`, etc) will typically be much easier to use in practice.

Usage

```
orderly_location_add(name, type, args, verify = TRUE, root = NULL)

orderly_location_add_path(name, path, verify = TRUE, root = NULL)

orderly_location_add_http(name, url, verify = TRUE, root = NULL)

orderly_location_add_packit(
  name,
  url,
  token = NULL,
  save_token = NULL,
  verify = TRUE,
  root = NULL
)
```

Arguments

`name`	The short name of the location to use. Cannot be in use, and cannot be one of `local` or `orphan`
`type`	The type of location to add. Currently supported values are `path` (a location that exists elsewhere on the filesystem), `http` (a location accessed over outpack's http API) and `packit` (a location accessed using the packit web app).

args	Arguments to the location driver. The arguments here will vary depending on the type used, see Details.
verify	Logical, indicating if we should verify that the location can be used before adding.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).
path	The path to the other archive root. This can be a relative or absolute path, with different tradeoffs. If you use an absolute path, then this location will typically work well on this machine, but it may behave poorly when the location is found on a shared drive **and** when you use your orderly root from more than one system. This setup is common when using an HPC system. If you use a relative path, then we will interpret it **relative to your orderly root** and not the directory that you evaluate this command from. Typically your path should include leading dots (e.g. ../../somewhere/else) as you should not nest orderly projects. This approach should work fine on shared filesystems.
url	The location of the server, including protocol, for example http://example.com:8080
token	The value for your your login token (currently this is a GitHub token with read:org scope). If NULL, orderly will perform an interactive authentication against GitHub to obtain one.
save_token	If no token is provided and interactive authentication is used, this controls whether the GitHub token should be saved to disk. Defaults to TRUE if NULL.

Details

We currently support three types of locations - path, which points to an outpack archive accessible by path (e.g., on the same computer or on a mounted network share), http, which requires that an outpack server is running at some url and uses an HTTP API to communicate, and packit, which uses Packit as a web server. More types may be added later, and more configuration options to these location types will definitely be needed in future.

Configuration options for different location types are described in the arguments to their higher-level functions.

Path locations:

Use orderly_location_add_path, which accepts a path argument.

HTTP locations:

Accessing outpack over HTTP requires that an outpack server is running. The interface here is expected to change as we expand the API, but also as we move to support things like TLS and authentication.

Use orderly_location_add_http, which accepts a url argument.

Packit locations:

Packit locations work over HTTPS, and include everything in an outpack location but also provide authentication and later will have more capabilities we think.

Use orderly_location_add_packit, which accepts url, token and save_token arguments.

Custom locations:

All outpack implementations are expected to support path and http locations, with the standard arguments above. But we expect that some implementations will support custom locations, and that the argument lists for these may vary between implementations. To allow this, you can pass a location of type "custom" with a list of arguments. We expect an argument 'driver' to be present among this list. For an example of this in action, see the `orderly.sharedfile` package.

Value

Nothing, called for the side effect of modifying the orderly configuration.

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()

# We create a packet in the remote root:
orderly_run("data", root = remote)

# Add the remote as a path location to the local root:
orderly_location_add_path("remote", remote, root = local)

# Pull metadata from 'remote' into our local version
orderly_location_fetch_metadata(root = local)

# Pull a packet into our local version
orderly_location_pull(quote(latest(name == "data")), root = local)

# Drop the location
orderly_location_remove("remote", root = local)
```

`orderly_location_fetch_metadata`

Fetch metadata from a location

Description

Fetch metadata from a location, updating the index. This should always be relatively quick as it updates only small files that contain information about what can be found in remote packets.

Usage

```
orderly_location_fetch_metadata(location = NULL, root = NULL)
```

Arguments

location	The name of a location to pull from (see orderly_location_list() for possible values). If not given, pulls from all locations. The "local" and "orphan" locations are always up to date and pulling metadata from them does nothing.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

Nothing

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()

# We create a packet in the remote root:
orderly_run("data", root = remote)

# Add the remote as a path location to the local root:
orderly_location_add_path("remote", remote, root = local)

# Pull metadata from 'remote' into our local version
orderly_location_fetch_metadata(root = local)
```

---

orderly_location_list *List known pack locations*

Description

List known locations. The special name local will always be present within the output from this function (this is packets known at the current root), though you will typically be interested in *other* locations.

Usage

```
orderly_location_list(verbose = FALSE, root = NULL)
```

Arguments

verbose	Logical, indicating if we should return a data.frame that includes more information about the location.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

Depending on the value of `verbose`:

- `verbose = FALSE`: A character vector of location names. This is the default behaviour.
- `verbose = TRUE`: A data.frame with columns `name`, `type` and `args`. The `args` column is a list column, with each element being the key-value pair arguments to the location.

See Also

`orderly_location_fetch_metadata()`, which can update your outpack index with metadata from any of the locations listed here.

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()

# No locations at first
orderly_location_list(root = local)

# Add a location
orderly_location_add_path("remote", remote, root = local)

# Here it is!
orderly_location_list(root = local)

# Add vebose = TRUE to find more about the location
orderly_location_list(root = local)
```

---

orderly_location_pull *Pull one or more packets from a location*

Description

Pull one or more packets (including all their files) into this archive from one or more of your locations. This will make files available for use as dependencies (e.g., with `orderly_dependency()`).

Usage

```
orderly_location_pull(
  expr,
  name = NULL,
  location = NULL,
  fetch_metadata = FALSE,
  recursive = NULL,
  options = NULL,
  root = NULL
)
```

Arguments

expr	The query expression. A NULL expression matches everything.
name	Optionally, the name of the packet to scope the query on. This will be intersected with scope arg and is a shorthand way of running scope = list(name = "name")
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
recursive	If non-NULL, a logical, indicating if we should recursively pull all packets that are referenced by the packets specified in id. This might copy a lot of data! If NULL, we default to the value given by the the configuration option require_complete_tree.
options	**DEPRECATED**. Please don't use this any more, and instead use the arguments location, allow_remote and fetch_metadata directly.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Details

It is possible that it will take a long time to pull packets, if you are moving a lot of data or if you are operating over a slow connection. Cancelling and resuming a pull should be fairly efficient, as we keep track of files that are copied over even in the case of an interrupted pull.

Value

Invisibly, the ids of packets that were pulled

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()

# We create a packet in the remote root:
orderly_run("data", root = remote)

# Add the remote as a path location to the local root:
orderly_location_add_path("remote", remote, root = local)

# Pull a packet into our local version
orderly_location_pull(quote(latest(name == "data")),
                      fetch_metadata = TRUE, root = local)
```

`orderly_location_push` *Push tree to location*

Description

Push tree to location. This function works out what packets are not known at the location and then what files are required to create them. It then pushes all the files required to build all packets and then pushes the missing metadata to the server. If the process is interrupted it is safe to resume and will only transfer files and packets that were missed on a previous call.

Usage

```
orderly_location_push(
  expr,
  location,
  name = NULL,
  dry_run = FALSE,
  root = NULL
)
```

Arguments

`expr`	An expression to search for. Often this will be a vector of ids, but you can use a query here.
`location`	The name of a location to push to (see `orderly_location_list()` for possible values).
`name`	Optionally, the name of the packet to scope the query on. This will be intersected with `scope` arg and is a shorthand way of running `scope = list(name = "name")`
`dry_run`	Logical, indicating if we should print a summary but not make any changes.
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Value

Invisibly, details on the information that was actually moved (which might be more or less than what was requested, depending on the dependencies of packets and what was already known on the other location).

Examples

```
# Two roots, one local and one representing some remote orderly
# location.  The remote location must use a file store at present.
local <- orderly_example()
```

```
remote <- orderly_example(use_file_store = TRUE)
orderly_location_add_path("remote", remote, root = local)

# We create a packet in the local root:
id <- orderly_run("data", root = local)

# Push a packet into our remote version
orderly_location_push(id, location = "remote", root = local)
```

`orderly_location_remove` *Remove a location*

Description

Remove an existing location. Any packets from this location and not known elsewhere will now be associated with the 'orphan' location instead.

Usage

```
orderly_location_remove(name, root = NULL)
```

Arguments

`name`	The short name of the location. Cannot remove `local` or `orphan`
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Value

Nothing

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()
orderly_location_add_path("remote", remote, root = local)

orderly_location_list(root = local)

# Remove the remote location:
orderly_location_remove("remote", root = local)
orderly_location_list(root = local)
```

`orderly_location_rename` *Rename a location*

Description

Rename an existing location

Usage

```
orderly_location_rename(old, new, root = NULL)
```

Arguments

`old`	The current short name of the location. Cannot rename `local` or `orphan`
`new`	The desired short name of the location. Cannot be one of `local` or `orphan`
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Value

Nothing

Examples

```
# Two roots, one local and one representing some remote orderly location:
local <- orderly_example()
remote <- orderly_example()
orderly_location_add_path("remote", remote, root = local)

orderly_location_list(root = local, verbose = TRUE)

# Rename the remote location:
orderly_location_rename("remote", "bob", root = local)
orderly_location_list(root = local, verbose = TRUE)
```

orderly_metadata *Read outpack metadata*

Description

Read metadata for a particular id. You may want to use `orderly_search()` to find an id corresponding to a particular query.

Usage

```
orderly_metadata(id, root = NULL)
```

Arguments

`id`	The id to fetch metadata for. An error will be thrown if this id is not known
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Value

A list of metadata. See the outpack schema for details (https://github.com/mrc-ide/outpack)

Examples

```
path <- orderly_example()
id <- orderly_run("data", root = path)

# Read metadata for this packet:
meta <- orderly_metadata(id, root = path)
names(meta)

# Information on files produced by this packet:
meta$files
```

orderly_metadata_extract *Extract metadata from orderly packets*

Description

Extract metadata from a group of packets. This is an **experimental** high-level function for interacting with the metadata in a way that we hope will be useful. We'll expand this a bit as time goes on, based on feedback we get so let us know what you think. See Details for how to use this.

Usage

```
orderly_metadata_extract(
  expr = NULL,
  name = NULL,
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE,
  extract = NULL,
  options = NULL,
  root = NULL
)
```

Arguments

expr	The query expression. A NULL expression matches everything.
name	Optionally, the name of the packet to scope the query on. This will be intersected with scope arg and is a shorthand way of running scope = list(name = "name")
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
extract	A character vector of columns to extract, possibly named. See Details for the format.
options	**DEPRECATED**. Please don't use this any more, and instead use the arguments location, allow_remote and fetch_metadata directly.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Details

Extracting data from outpack metadata is challenging to do in a way that works in data structures familiar to R users, because it is naturally tree structured, and because not all metadata may be present in all packets (e.g., a packet that does not depend on another will not have a dependency section, and one that was run in a context without git will not have git metadata). If you just want

the raw tree-structured data, you can always use `orderly_metadata()` to load the full metadata for any packet (even one that is not currently available on your computer, just known about it) and the structure of the data will remain fairly constant across orderly versions.

However, sometimes we want to extract data in order to ask specific questions like:

- what parameter combinations are available across a range of packets?
- when were a particular set of packets used?
- what files did these packets produce?

Later we'd like to ask even more complex questions like:

- at what version did the file `graph.png` change?
- what inputs changed between these versions?

...but being able to answer these questions requires a similar approach to interrogating metadata across a range of packets.

The `orderly_metadata_extract` function aims to simplify the process of pulling out bits of metadata and arranging it into a `data.frame` (of sorts) for you. It has a little mini-language in the `extract` argument for doing some simple rewriting of results, but you can always do this yourself.

In order to use function you need to know what metadata are available; we will expand the vignette with more worked examples here to make this easier to understand. The function works on top-level keys, of which there are:

- id: the packet id (this is always returned)
- name: the packet name
- parameters: a key-value pair of values, with string keys and atomic values. There is no guarantee about presence of keys between packets, or their types.
- time: a key-value pair of times, with string keys and time values (see DateTimeClasses; these are stored as seconds since 1970 in the actual metadata). At present `start` and `end` are always present.
- files: files present in each packet. This is a `data.frame` (per packet), each with columns `path` (relative), `size` (in bytes) and `hash`.
- depends: dependencies used each packet. This is a `data.frame` (per packet), each with columns `packet` (id), `query` (string, used to find `packet`) and `files` (another `data.frame` with columns `there` and `here` corresponding to filenames upstream and in this packet, respectively)
- git: either metadata about the state of git or `null`. If given then `sha` and `branch` are strings, while `url` is an array of strings/character vector (can have zero, one or more elements).
- session: some information about the session that the packet was run in (this is unstandardised, and even the orderly version may change)
- custom: additional metadata added by its respective engine. For packets run by `orderly`, there will be an `orderly` field here, which is itself a list:
  - artefacts: A data.frame with artefact information, containing columns `description` (a string) and `paths` (a list column of paths).
  - shared: A data.frame of the copied shared resources with their original name (`there`) and name as copied into the packet (`here`).

- role: A data.frame of identified roles of files, with columns `path` and `role`.
- description: A list of information from `orderly_description()` with human-readable descriptions and tags.
- session: A list of information about the session as run, with a list `platform` containing information about the platform (R version as `version`, operating system as `os` and system name as `system`) and `packages` containing columns `package` , `version` and `attached`.

The nesting here makes providing a universally useful data format difficult; if considering files we have a `data.frame` with a `files` column, which is a list of `data.frames`; similar nestedness applies to `depends` and the orderly custom data. However, you should be able to fairly easily process the data into the format you need it in.

The simplest extraction uses names of top-level keys:

```
extract = c("name", "parameters", "files")
```

This creates a data.frame with columns corresponding to these keys, one row per packet. Because `name` is always a string, it will be a character vector, but because `parameters` and `files` are more complex, these will be list columns.

You must not provide `id`; it is always returned and always first as a character vector column. If your extraction could possibly return data from locations (i.e., you have `allow_remote = TRUE` or have given a value for `location`) then we add a logical column `local` which indicates if the packet is local to your archive, meaning that you have all the files from it locally.

You can rename the columns by providing a name to entries within `extract`, for example:

```
extract = c("name", pars = "parameters", "files")
```

is the same as above, except that that the `parameters` column has been renamed `pars`.

More interestingly, we can index into a structure like `parameters`; suppose we want the value of the parameter `x`, we could write:

```
extract = c(x = "parameters.x")
```

which is allowed because for *each packet* the `parameters` element is a list.

However, we do not know what type `x` is (and it might vary between packets). We can add that information ourselves though and write:

```
extract = c(x = "parameters.x is number")
```

to create an numeric column. If any packet has a value of `x` that is non-integer, your call to `orderly_metadata_extract` will fail with an error, and if a packet lacks a value of `x`, a missing value of the appropriate type will be added.

Note that this does not do any coercion to number, it will error if a non-NULL non-numeric value is found. Valid types for use with `is <type>` are `boolean`, `number` and `string` (note that these differ slightly from R's names because we want to emphasise that these are *scalar* quantities; also note that there is no `integer` here as this may produce unexpected errors with integer-like numeric values). You can also use `list` but this is the default. Things in the schema that are known to be scalar atomics (such as `name`) will be automatically simplified.

You can index into the array-valued elements (`files` and `depends`) in the same way as for the object-valued elements:

```
extract = c(file_path = "files.path", file_hash = "files.hash")
```

would get you a list column of file names per packet and another of hashes, but this is probably less useful than the `data.frame` you'd get from extracting just `files` because you no longer have the hash information aligned.

You can index fairly deeply; it should be possible to get the orderly "display name" with:

```
extract = c(display = "custom.orderly.description.display is string")
```

If the path you need to extract has a dot in it (most likely a package name for a plugin, such as `custom.orderly.db`) you need to escape the dot with a backslash (so, `custom.orderly\.db`). You will probably need two slashes or use a raw string (in recent versions of R).

Value

A `data.frame`, the columns of which vary based on the names of `extract`; see Details for more information.

Custom 'orderly' metadata

Within `custom.orderly`, additional fields can be extracted. The format of this is subject to change, both in the stored metadata and schema (in the short term) and in the way we deserialise it. It is probably best not to rely on this right now, and we will expand this section when you can.

Examples

```
path <- orderly_example()

# Generate a bunch of packets:
suppressMessages({
  orderly_run("data", echo = FALSE, root = path)
  for (n in c(2, 4, 6, 8)) {
    orderly_run("parameters", list(max_cyl = n), echo = FALSE, root = path)
  }
})

# Without a query, we get a summary over all packets; this will
# often be too much:
orderly_metadata_extract(root = path)

# Pass in a query to limit things:
meta <- orderly_metadata_extract(quote(name == "parameters"), root = path)
meta

# The parameters are present as a list column:
meta$parameters

# You can also lift values from the parameters into columns of their own:
orderly_metadata_extract(
  quote(name == "parameters"),
  extract = c(max_cyl = "parameters.max_cyl is number"),
  root = path)
```

`orderly_metadata_read` *Read outpack metadata json file*

Description

Low-level function for reading metadata and deserialising it. This function can be used to directly read a metadata json file without reference to a root which contains it. It may be useful in the context of reading a metadata file written out as part of a failed run.

Usage

```
orderly_metadata_read(path, plugins = TRUE)
```

Arguments

`path`	Path to the json file
`plugins`	Try and deserialise data from all loaded plugins (see Details).

Details

Custom metadata saved by plugins may not be deserialised as expected when called with this function, as it is designed to operate separately from a valid orderly root (i.e., it will load data from any file regardless of where it came from). If `plugins` is `TRUE` (the default) then we will deserialise all data that matches any loaded plugin. This means that the behaviour of this function depends on if you have loaded the plugin packages. You can force this by running `orderly_config()` within any orderly directory, which will load any declared plugins.

Value

A list of outpack metadata; see the schema for details. In contrast to reading the json file directly with `jsonlite::fromJSON`, this function will take care to convert scalar and length-one vectors into the expected types.

Examples

```
path <- orderly_example()
id <- orderly_run("data", root = path)
meta <- orderly_metadata_read(file.path(path, ".outpack", "metadata", id))
identical(meta, orderly_metadata(id, root = path))
```

`orderly_migrate_source` *Migrate orderly source code*

Description

Migrate source code for an orderly project. Periodically, we may make changes to how orderly works that require you to update your source code sooner or later. This function can be used to automate (or at least accelerate) that process by trying to rewrite the R code within your project. See below for details of migrations and triggers for them.

Usage

```
orderly_migrate_source(path = ".", dry_run = FALSE, from = NULL, to = NULL)
```

Arguments

`path`	Path to the repository to migrate
`dry_run`	Logical, indicating if no changes would be made, but just print information about the changes that would be made. If `TRUE`, you can run this function against a repository that is not under version control.
`from`	Optional minimum version to migrate from. If `NULL`, we migrate from the version indicated in your orderly configuration and assume that all older migrations have been applied. You can specify a lower version here if you want to force migrations that would otherwise be skipped because they are assumed to be applied. Pass `"0"` (as a string) to match all previous versions.
`to`	Optional maximum version to migrate to. If `NULL` we apply all possible migrations. With `dry_run = TRUE` you may not want to use this, because we do not write any files, therefore each migration does not see the results of applying the previous migration.

Details

This function acts as an interface for rewriting the source code that will be used to create new packets, it does not migrate any data from packets that have been run. The idea here is that if we make changes to how orderly works that require some repetitive and relatively simple changes to your code, we can write a script that will do a reasonable (if not perfect) job of this, and you can run this over your code, check the results and if you like it commit the changes to your repository, rather than you having to go through and change everything by hand.

The version of orderly that you support is indicated by the version specified in `orderly_version.yml`; we will change some warnings to errors once you update this, in order to help you keep your code up to date.

Value

Primarily called for side effects, but returns (invisibly) `TRUE` if any changes were made (or would be made if `dry_run` was `TRUE`) and `FALSE` otherwise.

Migrations

A summary of migrations. The version number indicates the minimum version that this would increase your source repository to.

Currently, we do not *enforce* these changes must be present in a repository that declares it uses a recent orderly version, but this may happen at any time, without further warning!

1.99.82:

Removes references to `orderly2`, replacing them with `orderly`. This affects namespaced calls (e.g., `orderly2::orderly_parameter()`) and calls to `library` (e.g., `library(orderly2)`)

1.99.88:

Renames `<name>/orderly.R` files to `<name>/<name>.R`, a change that we introduced in early 2024 (version 1.99.13).

Future migrations:

We have some old changes to enable here:

- enforcing named arguments to `orderly_artefact`

We would like to enforce changes to `orderly_parameter` but have not worked out a general best practice way of doing this.

Migration process

This function requires a clean git status before it is run, and will typically be best to run against a fresh clone of a repository (though this is not enforced). After running, review changes (if any) with `git diff` and then commit. You cannot run this function against source code that is not version controlled with git.

We will refuse to migrate sources if we find the directories `archive/`, `draft/` or `.outpack/` to avoid any chance of modifying files in packets that have been previously run. You should make a fresh clone, migrate that, push back up to GitHub (or wherever you store your sources) and pull back down into your working directory.

Migration of very old sources

If you have old yaml-based orderly sources, you should consult `vignette("migrating")` as the migration path is not automatic and a bit more involved. You will need to install the helper package `outpack.orderly` and migrate your source and your archive separately.

Examples

```
# If a project already has made the migration from orderly2 to
# orderly, then the migration does nothing:
path <- orderly_example()
orderly_migrate_source(path, dry_run = TRUE)
```

`orderly_new` *Create a new report*

Description

Create a new empty report.

Usage

```
orderly_new(name, template = NULL, force = FALSE, root = NULL)
```

Arguments

`name`	The name of the report
`template`	The template to use. The only acceptable values for now are `NULL` (uses the built-in default) and `FALSE` which suppresses any default content. We may support customisable templates in future - let us know if this would be useful.
`force`	Create an orderly file - `<name>.R` within an existing directory `src/<name>`; this may be useful if you have already created the directory and some files first but want help creating the orderly file.
`root`	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function **does** require that the directory is configured for orderly, and not just outpack (see `orderly_init()` for details).

Value

Nothing, called for its side effects only

Examples

```
path <- withr::local_tempdir()

# Initialise a new repository, setting an option:
orderly_init(path)

# Create a new report 'myreport' in this root:
orderly_new("myreport", root = path)

# We now see:
fs::dir_tree(path, all = TRUE)

# By default, the new path will contain some hints, you can
# customise this by writing a template
cli::cli_code(readLines(file.path(path, "src", "myreport", "myreport.R")))
```

`orderly_parameters` *Declare orderly parameters*

Description

Declare orderly parameters. You should only have one call to this within your file! Typically you'd put the call to this function very close to the top of the file. Parameters are scalar atomic values (e.g. a string, number or boolean) and defaults must be present literally (i.e., they may not come from a variable itself). Provide `NULL` if you do not have a default, in which case this parameter will be required.

Usage

```
orderly_parameters(...)
```

Arguments

`...`	Any number of parameters. All arguments must be named.

Value

A list of parameters. This list is "strict" so accessing elements that are not present will throw an error rather than returning `NULL`.

Parameters and variables

Prior to orderly 1.99.61, parameters are always available as variables in the execution environment. In order to harmonise the R and Python versions of orderly, we are moving away from this, at least by default. The recommended way of using parameters is to assign it to a variable, for example:

```
pars <- orderly_parameters(debug = FALSE, replicates = NULL)
```

This defines two parameters, `debug` (with a default) and `replicates` (without a default). In the running report, you can access these by subsetting the `pars` object (e.g., `pars$debug` or `pars[["replicates"]]`).

To get the old behaviour, do not assign to a variable:

```
orderly_parameters(debug = FALSE, replicates = NULL)
```

This will create two bindings in the environment (`debug` and `replicates`) but will also generate a deprecation warning and we will remove support in a release of orderly 2.x. If you really want the old behaviour, you can achieve it by writing:

```
pars <- orderly_parameters(debug = FALSE, replicates = NULL)
list2env(pars, environment())
```

Behaviour in interactive sessions

We want you to be able to run through an orderly report interactively, e.g. via `source()`, by copy/paste or via the "Run" or "Source" button in RStudio. This is not very compatible with use of orderly parameters, because normally you'd provide these to `orderly_run()`, so we need a mechanism to get the parameters from you.

The behaviour differs if you have assigned the result of `orderly_parameters` to a variable or are using the (deprecated) behaviour of exporting parameters as variables.

New behaviour:

Suppose that you are assigning to `pars`. The first time we run though your code we won't see a value of `pars` and we'll prompt for values for each parameter. Those that have default values in your list will offer these values to make selection of parameters faster.

On subsequent calls, `pars` will be present with the values you used previously; these will be reused. If you want to be re-prompted, delete `pars` (i.e., `rm("pars")`) or assign `NULL` (i.e., `pars <- NULL`).

Old behaviour:

This is now deprecated, and you should update your code.

When running interactively (i.e., via `source()` or running an orderly file session by copy/paste or in RStudio), the `orderly_parameters()` function has different behaviour, and this behaviour depends on whether parameters will be exported to the environment or not.

First, we look in the current environment (most likely the global environment) for values of your parameters - that is, variables bound to the names of your parameters. For any parameters that are not found we will look at the default values and use these if possible, but if not possible then we will either error or prompt based on the global option `orderly.interactive_parameters_missing_error`. If this is `FALSE`, then we will ask you to enter a value for the parameters (strings will need to be entered with quotes).

Examples

```
# An example in context within the orderly examples, using the
# recommended new-style syntax:
orderly_example_show("parameters")
```

`orderly_parse_file` *Parse the orderly entrypoint script*

Description

For expert use only.

Usage

```
orderly_parse_file(path)

orderly_parse_expr(exprs, filename)
```

Arguments

path	Path to orderly_* script
exprs	Parsed AST from orderly_* script
filename	Name of orderly_* file to include in metadata

Details

Parses details of any calls to the orderly_ in-script functions into intermediate representation for downstream use. Also validates that any calls to orderly_* in-script functions are well-formed.

Value

Parsed orderly entrypoint script

Examples

```
path <- orderly_example()
# About the simplest case
orderly_parse_file(file.path(path, "src", "data", "data.R"))

# Find out about parameters
orderly_parse_file(file.path(path, "src", "parameters", "parameters.R"))

# Find out about dependencies:
orderly_parse_file(file.path(path, "src", "depends", "depends.R"))
```

---

orderly_plugin_add_metadata

Add metadata from plugin

Description

Add plugin-specific metadata to a running packet. This will take some describing. You accumulate any number of bits of metadata into arbitrary fields, and then later on serialise these to json.

Usage

```
orderly_plugin_add_metadata(name, field, data)
```

Arguments

name	The name of the plugin; must be the same as used in orderly_plugin_register() and orderly_plugin_context()
field	The name of a field to add the data to. This is required even if your plugin only produces one sort of data, in which case you can remove it later on within your serialisation function.
data	Arbitrary data to be added to the currently running packet

Value

Nothing, called only for its side effects

Examples

```
# The example code from vignette("plugins") is available in the package
fs::dir_tree(system.file("examples/example.db", package = "orderly"))

# See orderly_plugin_add_metadata in context here:
orderly_example_show("R/plugin.R", example = "example.db")
```

`orderly_plugin_context` *Fetch plugin context*

Description

Fetch the running context, for use within a plugin. The intention here is that within free functions that your plugin makes available, you will call this function to get information about the state of a packet. You will then typically call `orderly_plugin_add_metadata()` afterwards.

Usage

```
orderly_plugin_context(name, envir)
```

Arguments

`name`	Name of the plugin
`envir`	The environment of the calling function. You can typically pass `parent.frame()` (or `rlang::caller_env()`) here if the function calling `orderly_plugin_context()` is the function that would be called by a user. This argument only has an effect in interactive use (where `envir` is almost certainly the global environment).

Details

When a plugin function is called, orderly will be running in one of two modes; (1) from within `orderly_run()`, in which case we're part way through creating a packet in a brand new directory, and possibly using a special environment for evaluation, or (2) interactively, with a user developing their report. The plugin needs to be able to support both modes, and this function will return information about the state to help you cope with either case.

Value

A list with elements:

- `is_active`: a logical, indicating if we're running under `orderly_run()`; you may need to change behaviour depending on this value.

- path: the path of the running packet. This is almost always the working directory, unless the packet contains calls to setwd() or similar. You may create files here.
- config: the configuration for this plugin, after processing with the plugin's read function (see orderly_plugin_register)
- envir: the environment that the packet is running in. Often this will be the global environment, but do not assume this! You may read and write from this environment.
- src: the path to the packet source directory. This is different to the current directory when the packet is running, but the same when the user is interactively working with a report. You may *read* from this directory but *must not write to it*
- parameters: the parameters as passed through to the run the report.

See Also

orderly_plugin_register(), orderly_plugin_add_metadata()

Examples

```
# The example code from vignette("plugins") is available in the package
fs::dir_tree(system.file("examples/example.db", package = "orderly"))

# See orderly_plugin_context in context here:
orderly_example_show("R/plugin.R", example = "example.db")
```

---

orderly_plugin_register

Register an orderly plugin

Description

Create an orderly plugin. A plugin is typically defined by a package and is used to extend orderly by enabling new functionality, declared in your orderly configuration (orderly_config.json) and your orderly file (<name>.R), and affecting the running of reports primarily by creating new objects in the report environment. This system is discussed in more detail in vignette("plugins").

Usage

```
orderly_plugin_register(
  name,
  config,
  serialise = NULL,
  deserialise = NULL,
  cleanup = NULL,
  schema = NULL
)
```

Arguments

name	The name of the plugin, typically the package name
config	A function to read, check and process the configuration section in the orderly configuration. This function will be passed the deserialised data from the plugin's section of orderly_config.json, and the full path to that file. As the order of loading
serialise	A function to serialise any metadata added by the plugin's functions to the outpack metadata. It will be passed a list of all entries pushed in via orderly_plugin_add_metadata(); this is a named list with names corresponding to the field argument to orderly_plugin_add_metadata and each list element being an unnamed list with values corresponding to data. If NULL, then no serialisation is done, and no metadata from your plugin will be added.
deserialise	A function to deserialise any metadata serialised by the serialise function. This is intended to help deal with issues disambiguating unserialising objects from json (scalars vs arrays of length 1, data.frames vs lists-of-lists etc), and will make your plugin nicer to work with orderly_metadata_extract(). This function will be given a single argument data which is the data from jsonlite::fromJSON(..., simplifyVector = FALSE) and you should apply any required simplifications yourself, returning a modified copy of the argument.
cleanup	Optionally, a function to clean up any state that your plugin uses. You can call orderly_plugin_context from within this function and access anything you need from that. If not given, then no cleanup is done.
schema	Optionally a path, within the package, to a schema for the metadata created by this plugin; you should omit the .json extension. So if your file contains in its sources the file inst/plugin/myschema.json you would pass plugin/myschema. See vignette("plugins") for details.

Value

Nothing, this function is called for its side effect of registering a plugin.

Examples

```
# The example code from vignette("plugins") is available in the package
fs::dir_tree(system.file("examples/example.db", package = "orderly"))

# See orderly_plugin_register in context here:
orderly_example_show("R/plugin.R", example = "example.db")
```

---

orderly_prune_orphans *Prune orphan packet metadata*

Description

Prune orphan packets from your metadata store. This function can be used to remove references to packets that are no longer reachable; this could have happened because you deleted a packet manually from the archive and ran orderly_validate_archive() or because you removed a location.

Usage

```
orderly_prune_orphans(root = NULL)
```

Arguments

root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Details

If an orphan packet is not used anywhere, then we can easily drop it - it's as if it never existed. If it is referenced by metadata that you know about from elsewhere but not locally, then that is a problem for the upstream location (and one that should not happen). If you have referenced it in a packet that you have run locally, the the metadata is not deleted.

We expose this function mostly for users who want to expunge permanently any reference to previously run packets. We hope that there should never need to really be a reason to run it.

Value

Invisibly, a character vector of orphaned packet ids

Examples

```
# The same example as orderly_validate_archive; a corrupted
# archive due to the local deletion of a file
# Start with an archive containing 4 simple packets
path <- orderly_example()
ids <- vapply(1:4, function(i) orderly_run("data", root = path), "")
fs::file_delete(file.path(path, "archive", "data", ids[[3]], "data.rds"))

# Validate the archive ands orphan corrupt packets:
orderly_validate_archive(action = "orphan", root = path)

# Prune our orphans:
orderly_prune_orphans(root = path)
```

---

orderly_query *Construct outpack query*

Description

Construct an outpack query, typically then passed through to orderly_search()

Usage

```
orderly_query(expr, name = NULL, scope = NULL, subquery = NULL)
```

Arguments

expr	The query expression. A NULL expression matches everything.
name	Optionally, the name of the packet to scope the query on. This will be intersected with scope arg and is a shorthand way of running scope = list(name = "name")
scope	Optionally, a scope query to limit the packets searched by pars
subquery	Optionally, named list of subqueries which can be referenced by name from the expr.

Value

An orderly_query object, which should not be modified, but which can be passed to orderly_search()

See Also

vignette("dependencies") and vignette("query"), which discuss relationships between dependencies and the query DSL in more detail.

Examples

```
orderly_query(quote(latest(name == "data")))
```

---

orderly_query_explain *Explain a query*

Description

Explain how a query has or has not matched. This is experimental and the output will change. At the moment, it can tell you why a query matches, or if fails to match based on one of a number of &&-ed together clauses.

Usage

```
orderly_query_explain(
  expr,
  name = NULL,
  scope = NULL,
  subquery = NULL,
  parameters = NULL,
  envir = parent.frame(),
  location = NULL,
  allow_remote = NULL,
  root = NULL
)
```

Arguments

expr	The query expression. A NULL expression matches everything.
name	Optionally, the name of the packet to scope the query on. This will be intersected with scope arg and is a shorthand way of running scope = list(name = "name")
scope	Optionally, a scope query to limit the packets searched by pars
subquery	Optionally, named list of subqueries which can be referenced by name from the expr.
parameters	Optionally, a named list of parameters to substitute into the query (using the this: prefix)
envir	Optionally, an environment to substitute into the query (using the environment: prefix). The default here is to use the calling environment, but you can explicitly pass this in if you want to control where this lookup happens.
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
root	The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

An object of class orderly_query_explain, which can be inspected (contents subject to change) and which has a print method which will show a user-friendly summary of the query result.

Examples

```
path <- orderly_example()
suppressMessages({
  orderly_run("data", echo = FALSE, root = path)
  orderly_run("depends", echo = FALSE, root = path)
  for (n in c(2, 4, 6, 8)) {
    orderly_run("parameters", list(max_cyl = n), echo = FALSE, root = path)
  }
})

# Explain why a query matches some packets:
orderly_query_explain("parameter:max_cyl > 2 && name == 'parameters'",
                      root = path)

# Or misses:
```

```
orderly_query_explain("parameter:max_cyl > 2 && name == 'data'",
                      root = path)
```

`orderly_resource` *Declare orderly resources*

Description

Declare that a file, or group of files, are an orderly resource. By explicitly declaring files as resources orderly will mark the files as immutable inputs and validate that your analysis does not modify them when run with `orderly_run()`

Usage

```
orderly_resource(files)
```

Arguments

`files`	Any number of names of files or directories. If you list a directory it is expanded recursively to include all subdirectories and files.

Value

Invisibly, a character vector of resources included by the call. Don't rely on the order of these files if they are expanded from directories, as this is likely platform dependent. If a path was not found, then we throw an error.

Examples

```
# An example in context within the orderly examples:
orderly_example_show("strict")
```

`orderly_run` *Run a report*

Description

Run a report. This will create a new directory in `drafts/<reportname>`, copy your declared resources there, run your script and check that all expected artefacts were created.

Usage

```
orderly_run(
  name,
  parameters = NULL,
  envir = NULL,
  echo = TRUE,
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE,
  search_options = NULL,
  root = NULL
)
```

Arguments

name	Name of the report to run. Any leading `./` `src/` or trailing `/` path parts will be removed (e.g., if added by autocomplete).
parameters	Parameters passed to the report. A named list of parameters declared in the `orderly.yml`. Each parameter must be a scalar character, numeric, integer or logical.
envir	The environment that will be used to evaluate the report script; by default we use the global environment, which may not always be what is wanted.
echo	Optional logical to control printing output from `source()` to the console.
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is `TRUE`, then in conjunction with `orderly_dependency()` you might pull a large quantity of data. The default is `NULL`. This is `TRUE` if remote locations are listed explicitly as a character vector in the `location` argument, or if you have specified `fetch_metadata = TRUE`, otherwise `FALSE`.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If `location` is given, then we will pass this through to `orderly_location_fetch_metadata()` to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to `FALSE` after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
search_options	**DEPRECATED**. Please don't use this any more, and instead use the arguments `location`, `allow_remote` and `fetch_metadata` directly.
root	The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function **does** require that the directory is configured for orderly, and not just outpack (see `orderly_init()` for details).

Value

The id of the created report (a string)

Locations used in dependency resolution

If your packet depends on other packets, you will want to control the locations that are used to find appropriate packets. The control for this is passed through this function and *not* as an argument to `orderly_dependency()` because this is a property of the way that a packet is created and not of a packet itself; importantly different users may have different names for their locations so it makes little sense to encode the location name into the source code. Alternatively, you want to use different locations in different contexts, for example sometimes you might want to include local copies of packets as possible dependencies, but at other times you want to resolve dependencies only as they would be resolved on one of your locations.

Similarly, you might want to include packets that are known by other locations but are not currently downloaded onto this machine - pulling these packets in could take anything from seconds to hours depending on their size and the speed of your network connection (but *not* pulling in the packets could mean that your packet fails to run).

To allow for control over this you can pass in an arguments to control the names of the locations to use, whether metadata should be refreshed before we pull anything and if packets that are not currently downloaded should be considered candidates.

This has no effect when running interactively, in which case you can specify the search options (root specific) with `orderly_interactive_set_search_options()`

Which packets might be selected from locations?

The arguments `location`, `allow_remote` and `fetch_metadata` control where outpack searches for packets with the given query and if anything might be moved over the network (or from one outpack archive to another). By default everything is resolved locally only; that is we can only depend on packets that are unpacked within our current archive. If you pass `allow_remote = TRUE`, then packets that are known anywhere are candidates for using as dependencies and *if needed* we will pull the resolved files from a remote location. Note that even if the packet is not locally present this might not be needed - if you have the same content anywhere else in an unpacked packet we will reuse the same content without re-fetching.

If `fetch_metadata = TRUE`, then we will refresh location metadata before pulling, and the `location` argument controls which locations are pulled from.

Equivalence to the old `use_draft` option

The above location handling generalises the version 1.x of orderly's previous `use_draft` option, in terms of the new `location` argument:

- `use_draft = TRUE` is `location = "local"`
- `use_draft = FALSE` is `location = c(...)` where you should provide all locations *except* local (`setdiff(orderly_location_list(), "local")`)
- `use_draft = "newer"` is `location = NULL`

(this last option was the one most people preferred so is the new default behaviour). In addition, you could resolve dependencies as they currently exist on production right now with the options:

```
location = "production", fetch_metadata = TRUE
```

which updates your current metadata from production, then runs queries against only packets known on that remote, then depends on them even if you don't (yet) have them locally. This functionality was never available in orderly version 1, though we had intended to support it.

Running with a source tree separate from outpack root

Sometimes it is useful to run things from a different place on disk to your outpack root. We know of two cases where this has come up:

- when running reports within a runner on a server, we make a clean clone of the source tree at a particular git reference into a new temporary directory and then run the report there, but have it insert into an orderly repo at a fixed and non-temporary location.
- we have a user for whom it is more convenient to run their report on a hard drive but store the archive and metadata on a (larger) shared drive.

In the first instance, we have a source path at `<src>` which contains the file `orderly_config.json` and the directory `src/` with our source reports, and a separate path `<root>` which contains the directory `.outpack/` with all the metadata - it may also have an unpacked archive, and a `.git/` directory depending on the configuration. (Later this will make more sense once we support a "bare" outpack layout.)

Manually setting report source directory

To manually set the report source directory, you will need to set the path of the directory as the `ORDERLY_REPORT_SRC` environment variable.

Examples

```
# Create a simple example:
path <- orderly_example()

# Run the 'data' task:
orderly_run("data", root = path)

# After running, a finished packet appears in the archive:
fs::dir_tree(path)

# and we can query the metadata:
orderly_metadata_extract(name = "data", root = path)
```

---

orderly_run_info *Information about currently running report*

Description

Fetch information about the actively running report. This allows you to reflect information about your report back as part of the report, for example embedding the current report id, or information about computed dependencies. This information is in a slightly different format to orderly version 1.x and does not (currently) include information about dependencies when run outside of `orderly_run()`, but this was never reliable previously.

Usage

```
orderly_run_info()
```

Value

A list with elements

- name: The name of the current report
- id: The id of the current report, NA if running interactively
- root: The orderly root path
- depends: A data frame with information about the dependencies (not available interactively)
  - index: an integer sequence along calls to orderly_dependency()
  - name: the name of the dependency
  - query: the query used to find the dependency
  - id: the computed id of the included packet
  - filename: the file used from the packet
  - as: the filename used locally

Examples

```
# An example from the orderly examples
orderly_example_show("run_info")

# Prepare to run
path <- orderly_example()
orderly_run("data", root = path, echo = FALSE)

# Here, see the printed information from a real running report
orderly_run("run_info", root = path)
```

---

orderly_search *Query orderly's database*

Description

Evaluate a query against the orderly database (within .outpack/), returning a vector of matching packet ids. Note that by default this only searches through packets that are unpacked and available for direct use on this computer; to search within packets known to other locations (and that we might know about via their metadata) you will need to use the location, allow_remote and fetch_metadata arguments.

Usage

```
orderly_search(
  expr,
  name = NULL,
  scope = NULL,
  subquery = NULL,
  parameters = NULL,
  envir = parent.frame(),
  location = NULL,
  allow_remote = NULL,
  fetch_metadata = FALSE,
  options = NULL,
  root = NULL
)
```

Arguments

expr	The query expression. A NULL expression matches everything.
name	Optionally, the name of the packet to scope the query on. This will be intersected with scope arg and is a shorthand way of running scope = list(name = "name")
scope	Optionally, a scope query to limit the packets searched by pars
subquery	Optionally, named list of subqueries which can be referenced by name from the expr.
parameters	Optionally, a named list of parameters to substitute into the query (using the this: prefix)
envir	Optionally, an environment to substitute into the query (using the environment: prefix). The default here is to use the calling environment, but you can explicitly pass this in if you want to control where this lookup happens.
location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
fetch_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.
options	**DEPRECATED**. Please don't use this any more, and instead use the arguments location, allow_remote and fetch_metadata directly.

root — The path to the root directory, or NULL (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any outpack root (see orderly_init() for details).

Value

A character vector of matching ids. In the case of no match from a query returning a single value (e.g., latest(...) or single(...)) this will be a character missing value (NA_character_)

Examples

```
path <- orderly_example()

# Generate a bunch of packets:
suppressMessages({
  orderly_run("data", echo = FALSE, root = path)
  orderly_run("depends", echo = FALSE, root = path)
  for (n in c(2, 4, 6, 8)) {
    orderly_run("parameters", list(max_cyl = n), echo = FALSE, root = path)
  }
})

# By default, search returns everything, which is rarely what you want:
orderly_search(root = path)

# Restricting by name is common enough that there's a shortcut for
# it:
orderly_search(name = "data", root = path)
orderly_search(name = "parameters", root = path)

# Restrict to a parameter value:
orderly_search(quote(parameter:max_cyl > 4), name = "parameters",
               root = path)
```

---

orderly_search_options

Packet search options

Description

Options for controlling how packet searches are carried out, for example via orderly_search() and orderly_run(). The details here are never included in the metadata alongside the query (that is, they're not part of the query even though they affect it). (**This function is deprecated, please see below.**)

Usage

```
orderly_search_options(
  location = NULL,
  allow_remote = NULL,
  pull_metadata = FALSE
)
```

Arguments

location	Optional vector of locations to pull from. We might in future expand this to allow wildcards or exceptions.
allow_remote	Logical, indicating if we should allow packets to be found that are not currently unpacked (i.e., are known only to a location that we have metadata from). If this is TRUE, then in conjunction with orderly_dependency() you might pull a large quantity of data. The default is NULL. This is TRUE if remote locations are listed explicitly as a character vector in the location argument, or if you have specified fetch_metadata = TRUE, otherwise FALSE.
pull_metadata	Logical, indicating if we should pull metadata immediately before the search. If location is given, then we will pass this through to orderly_location_fetch_metadata() to filter locations to update. If pulling many packets in sequence, you *will* want to update this option to FALSE after the first pull, otherwise it will update the metadata between every packet, which will be needlessly slow.

Details

DEPRECATED: orderly_search() and orderly_run() now accept these arguments directly, which is much easier to reason about and use. A deprecation warning will be thrown by those functions if you pass options in.

Value

An object of class orderly_search_options which should not be modified after creation (but see note about fetch_metadata)

Examples

```
orderly_search_options()
```

---

orderly_shared_resource

Copy shared resources into a packet directory

Description

Copy shared resources into a packet directory. You can use this to share common resources (data or code) between multiple packets. Additional metadata will be added to keep track of where the files came from. Using this function requires the shared resources directory shared/ exists at the orderly root; an error will be raised if this is not configured when we attempt to fetch files.

Usage

```
orderly_shared_resource(...)
```

Arguments

`...`	The shared resources to copy. If arguments are named, the name will be the destination file while the value is the filename within the shared resource directory. You can use a limited form of string interpolation in the names of this argument; using `${variable}` will pick up values from `envir` and substitute them into your string. This is similar to the interpolation you might be familiar with from `glue::glue` or similar, but much simpler with no concatenation or other fancy features supported.

Value

Invisibly, a data.frame with columns `here` (the filenames as as copied into the running packet) and `there` (the filenames within `shared/`). Do not rely on the ordering where directory expansion was performed.

Examples

```
# An example in context within the orderly examples:
orderly_example_show("shared")

# Here's the directory structure for this example:
path <- orderly_example(names = "shared")
fs::dir_tree(path)

# We can run this packet:
orderly_run("shared", root = path)

# In the final archive version of the packet, 'cols.R' is copied
# over from `shared/`, so we have a copy of the version of the code
# that was used in the analysis
fs::dir_tree(path)
```

`orderly_strict_mode` *Enable orderly strict mode*

Description

Put orderly into "strict mode", which is closer to the defaults in orderly 1.0.0; in this mode only explicitly included files (via `orderly_resource()` and `orderly_shared_resource()`) are copied when running a packet, and we warn about any unexpected files at the end of the run. Using strict mode allows orderly to be more aggressive in how it deletes files within the source directory, more accurate in what it reports to you, and faster to start packets after developing them interactively.

Usage

```
orderly_strict_mode()
```

Details

In future, we may extend strict mode to allow requiring that no computation occurs within orderly functions (i.e., that the requirements to run a packet are fully known before actually running it). Most likely this will *not* be the default behaviour and `orderly_strict_mode()` will gain an argument.

We will allow server processes to either override this value (enabling it even when it is not explicitly given) and/or require it.

Value

Undefined

Examples

```
# An example in context within the orderly examples:
orderly_example_show("strict")
```

---

orderly_validate_archive

Validate unpacked packets.

Description

Validate unpacked packets. Over time, expect this function to become more fully featured, validating more.

Usage

```
orderly_validate_archive(
  expr = NULL,
  name = NULL,
  action = "inform",
  root = NULL
)
```

Arguments

`expr`	The query expression. A `NULL` expression matches everything.
`name`	Optionally, the name of the packet to scope the query on. This will be intersected with `scope` arg and is a shorthand way of running `scope = list(name = "name")`
`action`	The action to take on finding an invalid packet. See Details.

`root` The path to the root directory, or `NULL` (the default) to search for one from the current working directory. This function does not require that the directory is configured for orderly, and can be any `outpack` root (see `orderly_init()` for details).

Details

The actions that we can take on finding an invalid packet are:

- `inform` (the default): just print information about the problem
- `orphan`: mark the packet as orphaned within the metadata, but do not touch the files in your archive (by default the directory `archive/`) - this is a safe option and will leave you in a consistent state without deleting anything.
- `delete`: in addition to marking the packet as an orphan, also delete the files from your archive.

Later, we will add a "repair" option to try and fix broken packets.

The validation interacts with the option `core.require_complete_tree`; if this option is `TRUE`, then a packet is only valid if all its (recursive) dependencies are also valid, so the action will apply to packets that have also had their upstream dependencies invalidated. This validation will happen even if the query implied by `...` does not include these packets if a complete tree is required.

The validation will also interact with `core.use_file_store` once repair is supported, as this becomes trivial.

Value

Invisibly, a character vector of repaired (or invalid) packets.

Examples

```
# Start with an archive containing 4 simple packets
path <- orderly_example()
ids <- vapply(1:4, function(i) orderly_run("data", root = path), "")

# Suppose someone corrupts a packet by deleting a file:
fs::file_delete(file.path(path, "archive", "data", ids[[3]], "data.rds"))

# We can check all packets, and report on validity
orderly_validate_archive(root = path)

# Alternatively, we can take action and orphan the invalid packet:
orderly_validate_archive(action = "orphan", root = path)

# At which point the validation will not find this packet anymore
orderly_validate_archive(root = path)

# The orphaned packet will no longer be found in most operations:
orderly_search(root = path)
```

Index