Package 'pathling'

March 17, 2026

Type Package

Title A Library for using 'Pathling'

Version 9.5.0

Maintainer ``Australian e-Health Research Centre, CSIRO'' <pathling@csiro.au>

Description R API for 'Pathling', a tool for querying and transforming electronic health record data that is represented using the 'Fast Healthcare Interoperability Resources' (FHIR) standard - see <https://pathling.csiro.au/docs>.

License Apache License 2.0

URL https://pathling.csiro.au/

BugReports https://github.com/aehrc/pathling/issues

LazyData TRUE

Encoding UTF-8

Depends R (>= 3.5.0)

Imports rlang(>= 1.0.0), sparklyr(>= 1.8.1), jsonlite(>= 1.8.8), purrr

Suggests testthat(>= 3.2.1.1), lintr, styler

Config/testthat/edition 3

RoxygenNote 7.2.3

SystemRequirements Spark: 4.0.x

Config/pathling/Version 9.5.0

Config/pathling/SparkVersion 4.0.2

Config/pathling/ScalaVersion 2.13

Config/pathling/HadoopMajorVersion 3

Config/pathling/HadoopVersion 3.4.1

Config/pathling/DeltaVersion 4.0.0

NeedsCompilation yes

Author Australian e-Health Research Centre, CSIRO [cph, cre],
Piotr Szul [aut],
John Grimes [aut]

Repository CRAN

Date/Publication 2026-03-17 06:40:03 UTC

Contents

`conditions` *Synthetic conditions data*

Description

A synthetic data set of simplified and flattened FHIR Condition resources generated by Synthea.

Usage

```
conditions
```

Format

An object of class `data.frame` with 19 rows and 6 columns.

Details

A data frame with 19 rows and 6 columns:

- `START` - The onset date
- `STOP` - The abatement date
- `PATIENT` - The ID of the patient
- `ENCOUNTER` - The ID of the encounter
- `CODE` - The SNOMED CT code of the condition
- `DESCRIPTION` - The display name of the condition

`ds_read` *Get data for a resource type from a data source*

Description

Get data for a resource type from a data source

Usage

```
ds_read(ds, resource_code)
```

Arguments

`ds`	The DataSource object.
`resource_code`	A string representing the type of FHIR resource to read data from.

Value

A Spark DataFrame containing the data for the given resource type.

Examples

```
## Not run:
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()
data_source %>%
  ds_read("Condition") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

`ds_view` *Execute a SQL on FHIR view*

Description

Executes a SQL on FHIR view definition and returns the result as a Spark DataFrame.

Usage

```
ds_view(
  ds,
  resource,
  select = NULL,
  constants = NULL,
  where = NULL,
  json = NULL
)
```

Arguments

`ds`	The DataSource object containing the data to be queried.
`resource`	A string representing the type of FHIR resource that the view is based upon, e.g. 'Patient' or 'Observation'.
`select`	A list of columns and nested selects to include in the view. Each element should be a list with appropriate structure.
`constants`	An optional list of constants that can be used in FHIRPath expressions.
`where`	An optional list of FHIRPath expressions that can be used to filter the view.
`json`	An optional JSON string representing the view definition, as an alternative to providing the parameters as R objects.

Value

A Spark DataFrame containing the results of the view.

See Also

Pathling documentation - SQL on FHIR

Examples

```
## Not run:
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
data_source %>% ds_view("Patient",
  select = list(
    list(
      column = list(
        list(path = "id", name = "id"),
        list(path = "gender", name = "gender"),
        list(
          path = "telecom.where(system='phone').value",
          name = "phone_numbers", collection = TRUE
        )
      )
    ),
    list(
      forEach = "name",
      column = list(
        list(path = "use", name = "name_use"),
        list(path = "family", name = "family_name")
      ),
      select = list(
        list(
          forEachOrNull = "given",
          column = list(
            list(path = "$this", name = "given_name")
          )
        )
      )
    )
  ),
  where = list(
    list(path = "gender = 'male'")
  )
)

## End(Not run)
```

ds_write_delta *Write FHIR data to Delta files*

Description

Writes the data from a data source to a directory of Delta files.

Usage

```
ds_write_delta(ds, path, save_mode = SaveMode$OVERWRITE)
```

Arguments

`ds`	The DataSource object.
`path`	The URI of the directory to write the files to.
`save_mode`	The save mode to use when writing the data - "overwrite" will overwrite any existing data, "merge" will merge the new data with the existing data based on resource ID.

Value

A list with element `file_infos`, containing a list of files created. Each file has `fhir_resource_type` and `absolute_url`.

See Also

Pathling documentation - Writing Delta

`SaveMode`

Other data sink functions: `ds_write_ndjson()`, `ds_write_parquet()`, `ds_write_tables()`

Examples

```
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))

# Write the data to a directory of Delta files.
data_source %>% ds_write_delta(file.path(tempdir(), "delta"), save_mode = SaveMode$OVERWRITE)

pathling_disconnect(pc)
```

`ds_write_ndjson` *Write FHIR data to NDJSON files*

Description

Writes the data from a data source to a directory of NDJSON files. The files will be named using the resource type and the ".ndjson" extension.

Usage

```
ds_write_ndjson(ds, path, save_mode = SaveMode$ERROR, file_name_mapper = NULL)
```

Arguments

`ds`	The DataSource object.
`path`	The URI of the directory to write the files to.
`save_mode`	The save mode to use when writing the data.
`file_name_mapper`	An optional function that can be used to customise the mapping of the resource type to the file name. Currently not implemented.

Value

A list with element `file_infos`, containing a list of files created. Each file has `fhir_resource_type` and `absolute_url`.

See Also

Pathling documentation - Writing NDJSON

Other data sink functions: `ds_write_delta()`, `ds_write_parquet()`, `ds_write_tables()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))

# Write the data to a directory of NDJSON files.
data_source %>% ds_write_ndjson(file.path(tempdir(), "ndjson"))

## End(Not run)
```

---

ds_write_parquet *Write FHIR data to Parquet files*

Description

Writes the data from a data source to a directory of Parquet files.

Usage

```
ds_write_parquet(ds, path, save_mode = SaveMode$ERROR)
```

Arguments

`ds`	The DataSource object.
`path`	The URI of the directory to write the files to.
`save_mode`	The save mode to use when writing the data.

Value

A list with element `file_infos`, containing a list of files created. Each file has `fhir_resource_type` and `absolute_url`.

See Also

Pathling documentation - Writing Parquet

Other data sink functions: `ds_write_delta()`, `ds_write_ndjson()`, `ds_write_tables()`

Examples

```
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))

# Write the data to a directory of Parquet files.
data_source %>% ds_write_parquet(file.path(tempdir(), "parquet"))

pathling_disconnect(pc)
```

`ds_write_tables` *Write FHIR data to managed tables*

Description

Writes the data from a data source to a set of tables in the Spark catalog.

Usage

```
ds_write_tables(ds, schema = NULL, save_mode = SaveMode$OVERWRITE)
```

Arguments

`ds`	The DataSource object.
`schema`	The name of the schema to write the tables to.
`save_mode`	The save mode to use when writing the data - "overwrite" will overwrite any existing data, "merge" will merge the new data with the existing data based on resource ID.

Value

A list with element `file_infos`, containing a list of files created. Each file has `fhir_resource_type` and `absolute_url`.

See Also

Pathling documentation - Writing managed tables

`SaveMode`

Other data sink functions: `ds_write_delta`(), `ds_write_ndjson`(), `ds_write_parquet`()

Examples

```
# Create a temporary warehouse location, which will be used when we call ds_write_tables().
temp_dir_path <- tempfile()
dir.create(temp_dir_path)
sc <- sparklyr::spark_connect(master = "local[*]", config = list(
  "sparklyr.shell.conf" = c(
    paste0("spark.sql.warehouse.dir=", temp_dir_path),
    "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension",
    "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
  )
), version = pathling_spark_info()$spark_version)

pc <- pathling_connect(sc)
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))

# Write the data to a set of Spark tables in the 'default' database.
data_source %>% ds_write_tables("default", save_mode = SaveMode$MERGE)
```

```
pathling_disconnect(pc)
unlink(temp_dir_path, recursive = TRUE)
```

`Equivalence` *Concept map equivalence types*

Description

The following values are supported:

- `RELATEDTO` - The concepts are related to each other, and have at least some overlap in meaning, but the exact relationship is not known.
- `EQUIVALENT` - The definitions of the concepts mean the same thing (including when structural implications of meaning are considered) (i.e. extensionally identical).
- `EQUAL` - The definitions of the concepts are exactly the same (i.e. only grammatical differences) and structural implications of meaning are identical or irrelevant (i.e. intentionally identical).
- `WIDER` - The target mapping is wider in meaning than the source concept.
- `SUBSUMES` - The target mapping subsumes the meaning of the source concept (e.g. the source is-a target).
- `NARROWER` - The target mapping is narrower in meaning than the source concept. The sense in which the mapping is narrower SHALL be described in the comments in this case, and applications should be careful when attempting to use these mappings operationally.
- `SPECIALIZES` - The target mapping specializes the meaning of the source concept (e.g. the target is-a source).
- `INEXACT` - There is some similarity between the concepts, but the exact relationship is not known.
- `UNMATCHED` - This is an explicit assertion that there is no mapping between the source and target concept.
- `DISJOINT` - This is an explicit assertion that the target concept is not in any way related to the source concept.

Usage

```
Equivalence
```

Format

An object of class `list` of length 10.

See Also

FHIR R4 - ConceptMapEquivalence

`LOINC_URI` *LOINC system URI*

Description

The URI of the LOINC code system: `http://loinc.org`.

Usage

```
LOINC_URI
```

Format

An object of class `character` of length 1.

See Also

Using LOINC with HL7 Standards

`MimeType` *FHIR MIME types*

Description

The following MIME types are supported:

- `FHIR_JSON`: FHIR resources encoded as JSON
- `FHIR_XML`: FHIR resources encoded as XML

Usage

```
MimeType
```

Format

An object of class `list` of length 2.

See Also

FHIR R4 - Resource Formats

pathling_connect *Create or retrieve the Pathling context*

Description

Creates a Pathling context with the given configuration options.

Usage

```
pathling_connect(
  spark = NULL,
  max_nesting_level = 3,
  enable_extensions = FALSE,
 enabled_open_types = c("boolean", "code", "date", "dateTime", "decimal", "integer",
   "string", "Coding", "CodeableConcept", "Address", "Identifier", "Reference"),
  enable_terminology = TRUE,
  terminology_server_url = "https://tx.ontoserver.csiro.au/fhir",
  terminology_verbose_request_logging = FALSE,
  terminology_socket_timeout = 60000,
  max_connections_total = 32,
  max_connections_per_route = 16,
  terminology_retry_enabled = TRUE,
  terminology_retry_count = 2,
  enable_cache = TRUE,
  cache_max_entries = 2e+05,
  cache_storage_type = StorageType$MEMORY,
  cache_storage_path = NULL,
  cache_default_expiry = 600,
  cache_override_expiry = NULL,
  token_endpoint = NULL,
  enable_auth = FALSE,
  client_id = NULL,
  client_secret = NULL,
  scope = NULL,
  token_expiry_tolerance = 120,
  accept_language = NULL,
  explain_queries = FALSE,
  max_unbound_traversal_depth = 10
)
```

Arguments

`spark`	A pre-configured SparkSession instance, use this if you need to control the way that the session is set up
`max_nesting_level`	Controls the maximum depth of nested element data that is encoded upon import. This affects certain elements within FHIR resources that contain recursive

	references, e.g., QuestionnaireResponse.item.
`enable_extensions`	Enables support for FHIR extensions
`enabled_open_types`	The list of types that are encoded within open types, such as extensions.
`enable_terminology`	Enables the use of terminology functions
`terminology_server_url`	The endpoint of a FHIR terminology service (R4) that the server can use to resolve terminology queries.
`terminology_verbose_request_logging`	Setting this option to TRUE will enable additional logging of the details of requests to the terminology service.
`terminology_socket_timeout`	The maximum period (in milliseconds) that the server should wait for incoming data from the HTTP service
`max_connections_total`	The maximum total number of connections for the client
`max_connections_per_route`	The maximum number of connections per route for the client
`terminology_retry_enabled`	Controls whether terminology requests that fail for possibly transient reasons should be retried
`terminology_retry_count`	The number of times to retry failed terminology requests
`enable_cache`	Set this to FALSE to disable caching of terminology requests
`cache_max_entries`	Sets the maximum number of entries that will be held in memory
`cache_storage_type`	The type of storage to use for the terminology cache
`cache_storage_path`	The path on disk to use for the cache
`cache_default_expiry`	The default expiry time for cache entries (in seconds)
`cache_override_expiry`	If provided, this value overrides the expiry time provided by the terminology server
`token_endpoint`	An OAuth2 token endpoint for use with the client credentials grant
`enable_auth`	Enables authentication of requests to the terminology server
`client_id`	A client ID for use with the client credentials grant
`client_secret`	A client secret for use with the client credentials grant
`scope`	A scope value for use with the client credentials grant
`token_expiry_tolerance`	The minimum number of seconds that a token should have before expiry when deciding whether to send it with a terminology request

`accept_language`
: The default value of the Accept-Language HTTP header passed to the terminology server

`explain_queries`
: Setting this option to TRUE will enable additional logging relating to the query plan used to execute queries

`max_unbound_traversal_depth`
: Maximum depth for self-referencing structure traversals in repeat operations. Controls how deeply nested hierarchical data can be flattened during projection.

Details

If no Spark session is provided and there is not one already present in this process, a new one will be created.

If a SparkSession is not provided, and one is already running within the current process, it will be reused.

It is assumed that the Pathling library API JAR is already on the classpath. If you are running your own cluster, make sure it is on the list of packages.

Value

A Pathling context instance initialized with the specified configuration

See Also

Other context lifecycle functions: `pathling_disconnect()`, `pathling_disconnect_all()`, `pathling_spark()`

Examples

```
## Not run:
# Create PathlingContext for an existing Spark connecton.
pc <- pathling_connect(spark = sc)

# Create PathlingContext with a new Spark connection.
pc <- pathling_connect()
spark <- pathling_spark(pc)

## End(Not run)
```

`pathling_disconnect` *Disconnect from the Spark session*

Description

Disconnects the Spark connection associated with a Pathling context.

Usage

```
pathling_disconnect(pc)
```

Arguments

`pc`	The PathlingContext object.

Value

No return value, called for side effects only.

See Also

Other context lifecycle functions: `pathling_connect()`, `pathling_disconnect_all()`, `pathling_spark()`

`pathling_disconnect_all` *Disconnect all Spark connections*

Description

Disconnect all Spark connections

Usage

```
pathling_disconnect_all()
```

Value

No return value, called for side effects only.

See Also

Other context lifecycle functions: `pathling_connect()`, `pathling_disconnect()`, `pathling_spark()`

pathling_encode *Encode FHIR JSON or XML to a dataframe*

Description

Takes a Spark DataFrame with string representations of FHIR resources in the given column and encodes the resources of the given types as Spark DataFrame.

Usage

```
pathling_encode(pc, df, resource_name, input_type = NULL, column = NULL)
```

Arguments

`pc`	The Pathling context object.
`df`	A Spark DataFrame containing the resources to encode.
`resource_name`	The name of the FHIR resource to extract (e.g., "Condition", "Observation").
`input_type`	The MIME type of input string encoding. Defaults to "application/fhir+json".
`column`	The column in which the resources to encode are stored. If set to NULL, the input DataFrame is assumed to have one column of type string.

Value

A Spark DataFrame containing the given type of resources encoded into Spark columns.

See Also

Other encoding functions: `pathling_encode_bundle()`

Examples

```
pc <- pathling_connect()
json_resources_df <- pathling_spark(pc) %>%
  sparklyr::spark_read_text(path = system.file("extdata", "ndjson", "Condition.ndjson",
    package = "pathling"
  ))
pc %>% pathling_encode(json_resources_df, "Condition")
pathling_disconnect(pc)
```

`pathling_encode_bundle` *Encode FHIR Bundles to a dataframe*

Description

Takes a dataframe with string representations of FHIR bundles in the given column and outputs a dataframe of encoded resources.

Usage

```
pathling_encode_bundle(pc, df, resource_name, input_type = NULL, column = NULL)
```

Arguments

`pc`	A Pathling context object.
`df`	A Spark DataFrame containing the bundles with the resources to encode.
`resource_name`	The name of the FHIR resource to extract (Condition, Observation, etc.).
`input_type`	The MIME type of the input string encoding. Defaults to 'application/fhir+json'.
`column`	The column in which the resources to encode are stored. If 'NULL', then the input DataFrame is assumed to have one column of type string.

Value

A Spark DataFrame containing the given type of resources encoded into Spark columns.

See Also

Other encoding functions: `pathling_encode()`

Examples

```
pc <- pathling_connect()
json_resources_df <- pathling_spark(pc) %>%
  sparklyr::spark_read_text(
    path = system.file("extdata", "bundle-xml", package = "pathling"),
    whole = TRUE
  )
pc %>% pathling_encode_bundle(json_resources_df, "Condition",
  input_type = MimeType$FHIR_XML, column = "contents"
)
pathling_disconnect(pc)
```

pathling_evaluate_fhirpath

Evaluate a FHIRPath expression against a single FHIR resource

Description

Evaluates a FHIRPath expression against a single FHIR resource provided as a JSON string and returns materialised typed results. The resource is encoded into a one-row Spark Dataset internally, and the existing FHIRPath engine is used to evaluate the expression.

Usage

```
pathling_evaluate_fhirpath(
  pc,
  resource_type,
  resource_json,
  fhirpath_expression,
  context_expression = NULL,
  variables = NULL
)
```

Arguments

`pc`	The PathlingContext object.
`resource_type`	A string containing the FHIR resource type code (e.g., "Patient", "Observation").
`resource_json`	A string containing the FHIR resource as JSON.
`fhirpath_expression`	A FHIRPath expression to evaluate (e.g., "name.family", "gender = ’male’").
`context_expression`	An optional context expression string. If provided, the main expression is evaluated once for each result of the context expression. Defaults to NULL.
`variables`	An optional named list of variables available via `%variable` syntax. Defaults to NULL.

Value

A list with two elements:

`results` A list of lists, each containing `type` (character) and `value` (the materialised R value or NULL).

`expectedReturnType` A character string indicating the inferred return type.

See Also

Other context functions: `pathling_fhirpath_to_column()`, `pathling_filter()`, `pathling_search_to_column()`, `pathling_with_column()`

Examples

```
## Not run:
pc <- pathling_connect()
patient_json <- '{"resourceType": "Patient", "id": "example", "gender": "male"}'
result <- pathling_evaluate_fhirpath(pc, "Patient", patient_json, "gender")
for (entry in result$results) {
  cat(entry$type, ": ", entry$value, "\n")
}
pathling_disconnect(pc)

## End(Not run)
```

`pathling_examples` *Get path to Pathling example data*

Description

Construct the path to the package example data in a platform-independent way.

Usage

```
pathling_examples(...)
```

Arguments

`...`	character vector of the path components.

Value

The path to the examples data.

See Also

Other example functions: `pathling_example_resource()`

Examples

```
pathling_examples("ndjson", "Condition.ndjson")
```

`pathling_example_resource` *Read resource from Pathling example data*

Description

Reads a FHIR resource dataframe from the package example data.

Usage

```
pathling_example_resource(pc, resource_name)
```

Arguments

`pc`	The PathlingContext object.
`resource_name`	The name of the resource to read.

Details

The resources are read from the package example data in the `extdata/parquet` directory. Currently the following resources are available: 'Patient' and 'Condition'.

Value

A Spark DataFrame containing the resource data.

See Also

Other example functions: `pathling_examples()`

Examples

```
## Not run:
pathling_example_resource(pc, "Condition")

## End(Not run)
```

pathling_fhirpath_to_column *Convert a FHIRPath expression to a Spark Column*

Description

Converts a FHIRPath expression into a Spark Column that can be used in DataFrame operations such as filtering and selection. Boolean expressions can be used for filtering, while other expressions can be used for value extraction.

Usage

```
pathling_fhirpath_to_column(pc, resource_type, fhirpath_expression)
```

Arguments

`pc`	The PathlingContext object.
`resource_type`	A string containing the FHIR resource type code (e.g., "Patient", "Observation").
`fhirpath_expression`	A FHIRPath expression to evaluate (e.g., "gender = 'male'", "name.given.first()").

Details

The expression should evaluate to a single value per resource row.

Value

A Spark Column object (`spark_jobj`) representing the evaluated expression.

See Also

Other context functions: `pathling_evaluate_fhirpath()`, `pathling_filter()`, `pathling_search_to_column()`, `pathling_with_column()`

Examples

```
## Not run:
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
patients <- data_source %>% ds_read("Patient")

# Filter patients using a boolean FHIRPath expression.
filtered <- patients %>%
  pathling_filter(pc, "Patient", "gender = 'male'")

# Value expression for selection.
name_col <- pathling_fhirpath_to_column(pc, "Patient", "name.given.first()")
```

```
pathling_disconnect(pc)

## End(Not run)
```

---

pathling_filter *Filter a DataFrame using a FHIRPath or search expression*

Description

Filters a `tbl_spark` using either a FHIRPath boolean expression or a FHIR search query string, returning a `tbl_spark` containing only the matching rows. The DataFrame must be the first argument to enable piping with `%>%`.

Usage

```
pathling_filter(df, pc, resource_type, expression, type = "fhirpath")
```

Arguments

`df`	A `tbl_spark` containing FHIR resource data.
`pc`	The PathlingContext object.
`resource_type`	A string containing the FHIR resource type code (e.g., "Patient", "Observation").
`expression`	The filter expression. For `type = "fhirpath"`, a FHIRPath boolean expression (e.g., "gender = 'male'"). For `type = "search"`, a FHIR search query string (e.g., "gender=male&birthdate=ge1990-01-01").
`type`	The type of expression: `"fhirpath"` (default) or `"search"`.

Value

A `tbl_spark` containing only the rows matching the expression.

See Also

Other context functions: `pathling_evaluate_fhirpath()`, `pathling_fhirpath_to_column()`, `pathling_search_to_column()`, `pathling_with_column()`

Examples

```
## Not run:
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
patients <- data_source %>% ds_read("Patient")

# Filter using a FHIRPath expression.
male_patients <- patients %>%
  pathling_filter(pc, "Patient", "gender = 'male'")

# Filter using a FHIR search expression.
```

```
male_patients <- patients %>%
  pathling_filter(pc, "Patient", "gender=male", type = "search")

pathling_disconnect(pc)

## End(Not run)
```

`pathling_install_spark`
Install Spark

Description

Installs the version of Spark/Hadoop defined in the package metadata using the `sparklyr::spark_install` function.

Usage

```
pathling_install_spark()
```

Value

List with information about the installed version.

See Also

Other installation functions: `pathling_is_spark_installed()`, `pathling_spark_info()`, `pathling_version()`

`pathling_is_spark_installed`
Check if Spark is installed

Description

Checks if the version of Spark/Hadoop required by Pathling is installed.

Usage

```
pathling_is_spark_installed()
```

Value

`TRUE` if the required version of Spark/Hadoop is installed, `FALSE` otherwise.

See Also

Other installation functions: `pathling_install_spark()`, `pathling_spark_info()`, `pathling_version()`

pathling_read_bulk *Create a data source from a FHIR Bulk Data Access API endpoint*

Description

Creates a data source by downloading data from a FHIR server that implements the FHIR Bulk Data Access API.

Usage

```
pathling_read_bulk(
  pc,
  fhir_endpoint_url,
  output_dir,
  group_id = NULL,
  patients = NULL,
  types = NULL,
  output_format = "application/fhir+ndjson",
  since = NULL,
  elements = NULL,
  type_filters = NULL,
  include_associated_data = NULL,
  output_extension = "ndjson",
  timeout = NULL,
  max_concurrent_downloads = 10,
  auth_config = NULL
)
```

Arguments

`pc`	The PathlingContext object.
`fhir_endpoint_url`	The URL of the FHIR server to export from.
`output_dir`	The directory to write the output files to.
`group_id`	Optional group ID for group-level export.
`patients`	Optional list of patient IDs for patient-level export.
`types`	List of FHIR resource types to include.
`output_format`	The format of the output data. Defaults to "application/fhir+ndjson".
`since`	Only include resources modified after this timestamp.
`elements`	List of FHIR elements to include.
`type_filters`	FHIR search queries to filter resources.
`include_associated_data`	Pre-defined set of FHIR resources to include.

`output_extension`	File extension for output files. Defaults to "ndjson".
`timeout`	Optional timeout duration in seconds.
`max_concurrent_downloads`	Maximum number of concurrent downloads. Defaults to 10.
`auth_config`	Optional authentication configuration list with the following possible elements:

- enabled: Whether authentication is enabled (default: FALSE)
- client_id: The client ID to use for authentication
- private_key_jwk: The private key in JWK format
- client_secret: The client secret to use for authentication
- token_endpoint: The token endpoint URL
- use_smart: Whether to use SMART authentication (default: TRUE)
- use_form_for_basic_auth: Whether to use form-based basic auth (default: FALSE)
- scope: The scope to request
- token_expiry_tolerance: The token expiry tolerance in seconds (default: 120)

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading from Bulk Data API

Other data source functions: `pathling_read_bundles()`, `pathling_read_datasets()`, `pathling_read_delta()`, `pathling_read_ndjson()`, `pathling_read_parquet()`, `pathling_read_tables()`

Examples

```
## Not run:
pc <- pathling_connect()

# Basic system-level export
data_source <- pc %>% pathling_read_bulk(
  fhir_endpoint_url = "https://bulk-data.smarthealthit.org/fhir",
  output_dir = "/tmp/bulk_export"
)

# Group-level export with filters
data_source <- pc %>% pathling_read_bulk(
  fhir_endpoint_url = "https://bulk-data.smarthealthit.org/fhir",
  output_dir = "/tmp/bulk_export",
  group_id = "group-1",
  types = c("Patient", "Observation"),
  elements = c("id", "status"),
  since = as.POSIXct("2023-01-01")
)
```

```
# Patient-level export with auth
data_source <- pc %>% pathling_read_bulk(
  fhir_endpoint_url = "https://bulk-data.smarthealthit.org/fhir",
  output_dir = "/tmp/bulk_export",
  patients = c(
    "123", # Just the ID portion
    "456"
  ),
  auth_config = list(
    enabled = TRUE,
    client_id = "my-client-id",
    private_key_jwk = '{ "kty":"RSA", ...}',
    scope = "system/*.read"
  )
)

pathling_disconnect(pc)

## End(Not run)
```

`pathling_read_bundles` *Create a data source from FHIR bundles*

Description

Creates a data source from a directory containing FHIR bundles.

Usage

```
pathling_read_bundles(pc, path, resource_types, mime_type = MimeType$FHIR_JSON)
```

Arguments

`pc`	The PathlingContext object.
`path`	The URI of the directory containing the bundles.
`resource_types`	A sequence of resource type codes that should be extracted from the bundles.
`mime_type`	The MIME type of the bundles. Defaults to "application/fhir+json".

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading Bundles

Other data source functions: `pathling_read_bulk()`, `pathling_read_datasets()`, `pathling_read_delta()`, `pathling_read_ndjson()`, `pathling_read_parquet()`, `pathling_read_tables()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_bundles(
  pathling_examples("bundle-xml"),
  c("Patient", "Observation"), MimeType$FHIR_XML
)
data_source %>%
  ds_read("Observation") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

`pathling_read_datasets` *Create a data source from datasets*

Description

Creates an immutable, ad-hoc data source from a named list of Spark datasets indexed with resource type codes.

Usage

```
pathling_read_datasets(pc, resources)
```

Arguments

`pc`	The PathlingContext object.
`resources`	A name list of Spark datasets, where the keys are resource type codes and the values are the data frames containing the resource data.

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading datasets

Other data source functions: `pathling_read_bulk()`, `pathling_read_bundles()`, `pathling_read_delta()`, `pathling_read_ndjson()`, `pathling_read_parquet()`, `pathling_read_tables()`

Examples

```
## Not run:
patient_df <- pc %>% pathling_example_resource("Patient")
condition_df <- pc %>% pathling_example_resource("Condition")
data_source <- pc %>% pathling_read_datasets(list(Patient = patient_df, Condition = condition_df))
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

---

pathling_read_delta *Create a data source from Delta tables*

Description

`pathling_read_delta()` creates a data source from a directory containing Delta tables. Each table must be named according to the name of the resource type that it stores.

Usage

```
pathling_read_delta(pc, path)
```

Arguments

`pc`	The PathlingContext object.
`path`	The URI of the directory containing the Delta tables.

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading Delta

Other data source functions: `pathling_read_bulk()`, `pathling_read_bundles()`, `pathling_read_datasets()`, `pathling_read_ndjson()`, `pathling_read_parquet()`, `pathling_read_tables()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_delta(pathling_examples("delta"))
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

pathling_read_ndjson *Create a data source from NDJSON*

Description

Creates a data source from a directory containing NDJSON files. The files must be named with the resource type code and must have the ".ndjson" extension, e.g. "Patient.ndjson" or "Observation.ndjson".

Usage

```
pathling_read_ndjson(pc, path, extension = "ndjson", file_name_mapper = NULL)
```

Arguments

`pc`	The PathlingContext object.
`path`	The URI of the directory containing the NDJSON files.
`extension`	The file extension to use when searching for files. Defaults to "ndjson".
`file_name_mapper`	An optional function that maps a filename to the set of resource types that it contains. Currently not implemented.

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading NDJSON

Other data source functions: `pathling_read_bulk()`, `pathling_read_bundles()`, `pathling_read_datasets()`, `pathling_read_delta()`, `pathling_read_parquet()`, `pathling_read_tables()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

pathling_read_parquet *Create a data source from Parquet tables*

Description

`pathling_read_parquet()` creates a data source from a directory containing Parquet tables. Each table must be named according to the name of the resource type that it stores.

Usage

```
pathling_read_parquet(pc, path)
```

Arguments

`pc`	The PathlingContext object.
`path`	The URI of the directory containing the Parquet tables.

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading Parquet

Other data source functions: `pathling_read_bulk()`, `pathling_read_bundles()`, `pathling_read_datasets()`, `pathling_read_delta()`, `pathling_read_ndjson()`, `pathling_read_tables()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_parquet(pathling_examples("parquet"))
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

`pathling_read_tables` *Create a data source from managed tables*

Description

`pathling_read_tables()` creates a data source from a set of Spark tables, where the table names are the resource type codes.

Usage

```
pathling_read_tables(pc, schema = NULL)
```

Arguments

`pc`	The PathlingContext object.
`schema`	An optional schema name that should be used to qualify the table names.

Value

A DataSource object that can be used to run queries against the data.

See Also

Pathling documentation - Reading managed tables

Other data source functions: `pathling_read_bulk()`, `pathling_read_bundles()`, `pathling_read_datasets()`, `pathling_read_delta()`, `pathling_read_ndjson()`, `pathling_read_parquet()`

Examples

```
## Not run:
data_source <- pc %>% pathling_read_tables()
data_source %>%
  ds_read("Patient") %>%
  sparklyr::sdf_nrow()

## End(Not run)
```

pathling_search_to_column

Convert a FHIR search expression to a Spark Column

Description

Converts a FHIR search query string into a Spark Column representing a boolean filter condition. The returned Column can be used with sparklyr DataFrame operations such as `sdf_filter` to filter resources matching the search criteria.

Usage

```
pathling_search_to_column(pc, resource_type, search_expression)
```

Arguments

`pc`	The PathlingContext object.
`resource_type`	A string containing the FHIR resource type code (e.g., "Patient", "Observation").
`search_expression`	A FHIR search query string in URL query format (e.g., "gender=male&birthdate=ge1990-01-01"). An empty string matches all resources.

Value

A Spark Column object (`spark_jobj`) representing the boolean filter condition.

See Also

Other context functions: `pathling_evaluate_fhirpath()`, `pathling_fhirpath_to_column()`, `pathling_filter()`, `pathling_with_column()`

Examples

```
## Not run:
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
patients <- data_source %>% ds_read("Patient")

# Filter patients by gender.
filtered <- patients %>%
  pathling_filter(pc, "Patient", "gender=male", type = "search")

# Multiple search parameters (AND).
filtered <- patients %>%
  pathling_filter(pc, "Patient", "gender=male&active=true", type = "search")

pathling_disconnect(pc)

## End(Not run)
```

`pathling_spark` *Get the Spark session*

Description

Returns the Spark connection associated with a Pathling context.

Usage

```
pathling_spark(pc)
```

Arguments

`pc`	The PathlingContext object.

Value

The Spark connection associated with this Pathling context.

See Also

Other context lifecycle functions: `pathling_connect()`, `pathling_disconnect()`, `pathling_disconnect_all()`

`pathling_spark_info` *Get versions of Spark and other dependencies*

Description

Returns the versions of Spark and Spark packages used by the Pathling R library.

Usage

```
pathling_spark_info()
```

Value

A list containing the following keys:

- `spark_version`: The version of Spark used by Pathling.
- `scala_version`: The version of Scala used by Pathling.
- `hadoop_version`: The version of Hadoop used by Pathling.
- `hadoop_major_version`: The major version of Hadoop used by Pathling.
- `delta_version`: The version of Delta used by Pathling.

See Also

Other installation functions: `pathling_install_spark()`, `pathling_is_spark_installed()`, `pathling_version()`

`pathling_version` *Get version of Pathling*

Description

Get version of Pathling

Usage

```
pathling_version()
```

Value

The version of the Pathling R library.

See Also

Other installation functions: `pathling_install_spark()`, `pathling_is_spark_installed()`, `pathling_spark_info()`

`pathling_with_column` *Add a FHIRPath-derived column to a DataFrame*

Description

Evaluates a FHIRPath expression and adds the result as a named column to a `tbl_spark`, returning the augmented `tbl_spark`. The DataFrame must be the first argument to enable piping with `%>%`. Multiple calls can be chained to add several columns.

Usage

```
pathling_with_column(df, pc, resource_type, expression, column)
```

Arguments

`df`	A `tbl_spark` containing FHIR resource data.
`pc`	The PathlingContext object.
`resource_type`	A string containing the FHIR resource type code (e.g., "Patient", "Observation").
`expression`	A FHIRPath expression to evaluate (e.g., "name.given.first()").
`column`	The name of the new column to add.

Value

A `tbl_spark` with the new column added.

See Also

Other context functions: `pathling_evaluate_fhirpath()`, `pathling_fhirpath_to_column()`, `pathling_filter()`, `pathling_search_to_column()`

Examples

```
## Not run:
pc <- pathling_connect()
data_source <- pc %>% pathling_read_ndjson(pathling_examples("ndjson"))
patients <- data_source %>% ds_read("Patient")

# Add a single column.
result <- patients %>%
  pathling_with_column(pc, "Patient", "name.given.first()", column = "given_name")

# Chain multiple columns.
result <- patients %>%
  pathling_with_column(pc, "Patient", "name.given.first()", column = "given_name") %>%
  pathling_with_column(pc, "Patient", "gender", column = "gender_value") %>%
  dplyr::select(id, given_name, gender_value)

pathling_disconnect(pc)

## End(Not run)
```

`PropertyType` *Coding property data types*

Description

The following data types are supported:

- `STRING` - A string value.
- `INTEGER` - An integer value.
- `BOOLEAN` - A boolean value.
- `DECIMAL` - A decimal value.
- `DATETIME` - A date/time value.
- `CODE` - A code value.
- `CODING` - A Coding value.

Usage

```
PropertyType
```

Format

An object of class `list` of length 7.

See Also

FHIR R4 - Data Types

SaveMode *SaveMode*

Description

The following save modes are supported:

- `OVERWRITE`: Overwrite any existing data.
- `APPEND`: Append the new data to the existing data.
- `IGNORE`: Only save the data if the file does not already exist.
- `ERROR`: Raise an error if the file already exists.
- `MERGE`: Merge the new data with the existing data based on resource ID.

Usage

```
SaveMode
```

Format

An object of class `list` of length 5.

SNOMED_URI *SNOMED CT system URI*

Description

The URI of the SNOMED CT code system: `http://snomed.info/sct`.

Usage

```
SNOMED_URI
```

Format

An object of class `character` of length 1.

See Also

Using SNOMED CT with HL7 Standards

`StorageType` *Terminology cache storage type*

Description

The type of storage to use for the terminology cache.

Usage

```
StorageType
```

Format

An object of class `list` of length 2.

Details

The following values are supported:

- `MEMORY` - Use an in-memory cache
- `DISK` - Use a disk-based cache

`to_array` *Convert a vector to a SQL array literal*

Description

Converts a vector to an expression with the corresponding SQL array literal.

Usage

```
to_array(value)
```

Arguments

`value`	A character or numeric vector to be converted

Value

The `quosure` with the SQL array literal that can be used in `dplyr::mutate`.

tx_designation *Get designations for codings*

Description

Takes a Coding column as its input. Returns a Column that contains the values of designations (strings) for this coding that match the specified use and language. If the language is not provided, then all designations with the specified type are returned regardless of their language.

Usage

```
tx_designation(coding, use = NULL, language = NULL)
```

Arguments

`coding`	A Column containing a struct representation of a Coding.
`use`	The code with the use of the designations.
`language`	The language of the designations.

Value

The Column containing the result of the operation (array of strings with designation values).

See Also

Pathling documentation - Retrieving designations

Examples

```
## Not run:
# Get the (first) SNOMED CT "Fully specified name" ('900000000000003001')
# for the first coding of the Condition resource, in the 'en' language.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(
    id,
    designation = (!!tx_designation(code[["coding"]][[0]],
      !!tx_to_snomed_coding("900000000000003001"),
      language = "en"
    ))[[0]],
    .keep = "none"
  )

## End(Not run)
```

tx_display *Get the display text for codings*

Description

Takes a Coding column as its input. Returns a Column that contains the canonical display name associated with the given code.

Usage

```
tx_display(coding, accept_language = NULL)
```

Arguments

`coding`	A Column containing a struct representation of a Coding.
`accept_language`	The optional language preferences for the returned display name. Overrides the parameter ‘accept_language‘ in `pathling_connect`.

Value

A Column containing the result of the operation (String).

See Also

Pathling documentation - Multi-language support

Other terminology functions: `tx_member_of()`, `tx_property_of()`, `tx_subsumed_by()`, `tx_subsumes()`, `tx_translate()`

Examples

```
## Not run:
# Get the display name of the first coding of the Condition resource, with the default language.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(
    id,
    display = !!tx_display(code[["coding"]][[0]]),
    .keep = "none"
  )

## End(Not run)
```

`tx_member_of` *Test membership within a value set*

Description

Takes a Coding or array of Codings column as its input. Returns the column which contains a Boolean value, indicating whether any of the input Codings is a member of the specified FHIR ValueSet.

Usage

```
tx_member_of(codings, value_set_uri)
```

Arguments

`codings`	A Column containing a struct representation of a Coding or an array of such structs.
`value_set_uri`	An identifier for a FHIR ValueSet.

Value

A Column containing the result of the operation.

See Also

Pathling documentation - Value set membership

Other terminology functions: `tx_display()`, `tx_property_of()`, `tx_subsumed_by()`, `tx_subsumes()`, `tx_translate()`

`tx_property_of` *Get properties for codings*

Description

Takes a Coding column as its input. Returns a Column that contains the values of properties for this coding with specified names and types. The type of the result column depends on the types of the properties. Primitive FHIR types are mapped to their corresponding SQL primitives. Complex types are mapped to their corresponding structs.

Usage

```
tx_property_of(
  coding,
  property_code,
  property_type = "string",
  accept_language = NULL
)
```

Arguments

`coding`	A Column containing a struct representation of a Coding.
`property_code`	The code of the property to retrieve.
`property_type`	The type of the property to retrieve.
`accept_language`	The optional language preferences for the returned property values. Overrides the parameter ‘accept_language‘ in ‘PathlingContext.create‘.

Value

The Column containing the result of the operation (array of property values).

See Also

`PropertyType`

Pathling documentation - Retrieving properties

Other terminology functions: `tx_display()`, `tx_member_of()`, `tx_subsumed_by()`, `tx_subsumes()`, `tx_translate()`

Examples

```
## Not run:
# Get the (first) value of the `inactive` property of the first coding of the Condition resource.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(id,
    is_inavtive = (!!tx_property_of(
      code[["coding"]][[0]],
      "inactive", PropertyType$BOOLEAN
    ))[[0]],
    .keep = "none"
  )

## End(Not run)
```

`tx_subsumed_by` *Test subsumption between codings*

Description

Takes two Coding columns as input. Returns a Column that contains a Boolean value, indicating whether the left Coding is subsumed by the right Coding.

Usage

```
tx_subsumed_by(left_codings, right_codings)
```

Arguments

`left_codings`	A Column containing a struct representation of a Coding or an array of Codings.
`right_codings`	A Column containing a struct representation of a Coding or an array of Codings.

Value

A Column containing the result of the operation (boolean).

See Also

Pathling documentation - Subsumption testing

Other terminology functions: `tx_display()`, `tx_member_of()`, `tx_property_of()`, `tx_subsumes()`, `tx_translate()`

Examples

```
pc <- pathling_connect()

# Test the codings of the Condition `code` for subsumption by a SNOMED CT code.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(
    id,
    is_subsumed_by = !!tx_subsumed_by(
      code[["coding"]],
      !!tx_to_snomed_coding("444814009")
    ),
    .keep = "none"
  )

pathling_disconnect(pc)
```

---

tx_subsumes *Test subsumption between codings*

Description

Takes two Coding columns as input. Returns a Column that contains a Boolean value, indicating whether the left Coding subsumes the right Coding.

Usage

```
tx_subsumes(left_codings, right_codings)
```

Arguments

`left_codings`	A Column containing a struct representation of a Coding or an array of Codings.
`right_codings`	A Column containing a struct representation of a Coding or an array of Codings.

Value

A Column containing the result of the operation (boolean).

See Also

Pathling documentation - Subsumption testing

Other terminology functions: `tx_display()`, `tx_member_of()`, `tx_property_of()`, `tx_subsumed_by()`, `tx_translate()`

Examples

```
## Not run:
# Test the codings of the Condition `code` for subsumption of a SNOMED CT code.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(
    id,
    subsumes = !!tx_subsumes(
      code[["coding"]],
      !!tx_to_snomed_coding("444814009")
    ),
    .keep = "none"
  )

## End(Not run)
```

`tx_to_coding` *Convert codes to Coding structures*

Description

Converts a Column containing codes into a Column that contains a Coding struct.

Usage

```
tx_to_coding(coding_column, system, version = NULL)
```

Arguments

`coding_column`	The Column containing the codes.
`system`	The URI of the system the codes belong to.
`version`	The version of the code system.

Details

The Coding struct Column can be used as an input to terminology functions such as `tx_member_of` and `tx_translate`. Please note that inside `sparklyr` verbs such as `mutate` the functions calls need to be preceded with `!!`, e.g: `!!tx_to_coding(CODE, SNOMED_URI)`.

Value

A Column containing a Coding struct.

See Also

FHIR R4 - Coding

Other terminology helpers: `tx_to_ecl_value_set()`, `tx_to_loinc_coding()`, `tx_to_snomed_coding()`

Examples

```
## Not run:
condition_df <- pathling_spark(pc) %>% sparklyr::copy_to(conditions)

# Convert codes to ICD-10 codings.
condition_df %>% sparklyr::mutate(
  icdCoding = !!tx_to_coding(CODE, "http://hl7.org/fhir/sid/icd-10"), .keep = "none"
)

## End(Not run)
```

`tx_to_ecl_value_set` *Convert a SNOMED CT ECL expression to a ValueSet URI*

Description

Converts a SNOMED CT ECL expression into a FHIR ValueSet URI. It can be used with the ‘`tx_member_of` function.

Usage

```
tx_to_ecl_value_set(ecl)
```

Arguments

`ecl`	The ECL expression.

Value

The ValueSet URI.

See Also

Using SNOMED CT with HL7 Standards - Implicit Value Sets

Other terminology helpers: `tx_to_coding()`, `tx_to_loinc_coding()`, `tx_to_snomed_coding()`

Examples

```
## Not run:
# Example usage of tx_to_ecl_value_set function
tx_to_ecl_value_set("<<373265006 |Analgesic (substance)|")

## End(Not run)
```

`tx_to_loinc_coding` *Convert LOINC codes to Coding structures*

Description

Converts a Column containing codes into a Column that contains a LOINC Coding struct.

Usage

```
tx_to_loinc_coding(coding_column, version = NULL)
```

Arguments

`coding_column`	The Column containing the codes.
`version`	The version of the code system.

Details

The Coding struct Column can be used as an input to terminology functions such as `tx_member_of` and `tx_translate`. Please note that inside `sparklyr` verbs such as `mutate` the functions calls need to be preceded with `!!`, e.g: `!!tx_to_coding(CODE, SNOMED_URI)`.

Value

A Column containing a Coding struct.

See Also

Other terminology helpers: `tx_to_coding()`, `tx_to_ecl_value_set()`, `tx_to_snomed_coding()`

Examples

```
## Not run:
condition_df <- pathling_spark(pc) %>% sparklyr::copy_to(conditions)

# Convert codes to LOINC codings.
# Equivalent to: tx_to_coding(CODE, "http://loinc.org")
condition_df %>% sparklyr::mutate(loincCoding = !!tx_to_loinc_coding(CODE), .keep = "none")

## End(Not run)
```

tx_to_snomed_coding *Convert SNOMED CT codes to Coding structures*

Description

Converts a Column containing codes into a Column that contains a SNOMED Coding struct.

Usage

```
tx_to_snomed_coding(coding_column, version = NULL)
```

Arguments

`coding_column`	The Column containing the codes.
`version`	The version of the code system.

Details

The Coding struct Column can be used as an input to terminology functions such as `tx_member_of` and `tx_translate`. Please note that inside `sparklyr` verbs such as `mutate` the functions calls need to be preceded with `!!`, e.g: `!!tx_to_coding(CODE, SNOMED_URI)`.

Value

A Column containing a Coding struct.

See Also

Other terminology helpers: `tx_to_coding()`, `tx_to_ecl_value_set()`, `tx_to_loinc_coding()`

Examples

```
## Not run:
condition_df <- pathling_spark(pc) %>% sparklyr::copy_to(conditions)

# Convert codes to SNOMED CT codings.
# Equivalent to: tx_to_coding(CODE, "http://snomed.info/sct")
condition_df %>% sparklyr::mutate(snomedCoding = !!tx_to_snomed_coding(CODE), .keep = "none")

## End(Not run)
```

`tx_translate` *Translate between value sets*

Description

Takes a Coding column as input. Returns the Column which contains an array of Coding value with translation targets from the specified FHIR ConceptMap. There may be more than one target concept for each input concept. Only the translation with the specified equivalences are returned.

Usage

```
tx_translate(
  codings,
  concept_map_uri,
  reverse = FALSE,
  equivalences = NULL,
  target = NULL
)
```

Arguments

`codings`	A Column containing a struct representation of a Coding.
`concept_map_uri`	An identifier for a FHIR ConceptMap.
`reverse`	The direction to traverse the map. FALSE results in "source to target" mappings, while TRUE results in "target to source".
`equivalences`	A value of a collection of values from the ConceptMapEquivalence ValueSet.
`target`	Identifies the value set in which a translation is sought. If there's no target specified, the server should return all known translations.

Value

A Column containing the result of the operation (an array of Coding structs).

See Also

`Equivalence`

Pathling documentation - Concept translation

Other terminology functions: `tx_display()`, `tx_member_of()`, `tx_property_of()`, `tx_subsumed_by()`, `tx_subsumes()`

Examples

```
## Not run:
# Translates the codings of the Condition `code` using a SNOMED implicit concept map.
pc %>%
  pathling_example_resource("Condition") %>%
  sparklyr::mutate(
    id,
    translation = !!tx_translate(
      code[["coding"]],
      "http://snomed.info/sct?fhir_cm=900000000000527005"
    ),
    .keep = "none"
  )

## End(Not run)
```

`Version` *FHIR versions*

Description

The following FHIR versions are supported:

- `R4`: FHIR R4

Usage

```
Version
```

Format

An object of class `list` of length 1.

Index