

Package: synapser (via r-universe)

October 31, 2026

Type Package

Title R Language Bindings for Synapse API

Version 3.0.0

Date 2026-10-01

Description Provides R language bindings for the 'Synapse' platform <<https://www.synapse.org>>, a collaborative research platform from Sage Bionetworks for sharing data, tracking analyses, and managing provenance. Supports authentication, uploading and downloading files, managing projects, folders, tables, views, annotations, and access permissions. Built on the official 'Synapse' 'Python' client ('synapseclient') via the 'reticulate' package.

Encoding UTF-8

License Apache License 2.0

Imports reticulate (>= 1.44.0), methods, rjson, stats, utils

Depends R(>= 4.2.0)

Suggests testthat, knitr, rmarkdown, tibble, pkgdown (>= 2.2.0)

URL <https://github.com/Sage-Bionetworks/synapser>,
<https://r-docs.synapse.org>, <https://www.synapse.org>

BugReports <https://sagebionetworks.jira.com/servicedesk/customer/portal/5/group/7/create/144>

VignetteBuilder knitr

SystemRequirements synapseclient
(<https://python-docs.synapse.org/en/stable/>)

RoxygenNote 7.2.3

Config/pak/sysreqs libpng-dev python3

Repository <https://sage-bionetworks.r-universe.dev>

Date/Publication 2026-10-31 00:41:33 UTC

RemoteUrl <https://github.com/Sage-Bionetworks/synapser>

RemoteRef SYN-1649-upgrade-pythonclient-4-14

RemoteSha c74ced761d3f74cd148e97b1e09fb78af2727132

Contents

Activity	7
ActivityOptions	9
Agent	10
AgentPrompt	13
AgentSession	14
Column	16
Dataset	18
DatasetCollection	25
DownloadListItem	31
EntityRef	32
EntityView	33
Evaluation	40
File	43
FileOptions	47
Folder	49
Link	54
LinkOptions	55
MaterializedView	56
Organization	62
Project	63
StorageLocation	68
StorageLocationType	71
StoreContainerOptions	72
StoreFileOptions	73
StoreGridOptions	74
StoreJSONSchemaOptions	75
StoreTableOptions	76
Submission	77
SubmissionBundle	79
SubmissionStatus	80
SubmissionView	83
synAddColumn	88
synAddColumn	90
synAddColumn	91
synAddColumn	93
synAddColumn	94
synAddItem	96
synAddItem	97
synBatchUpdateSubmissionStatuses	98
synBindSchema	99
synBindSchema	101
synBindSchema	102
synBindSchema	104
synBindSchema	106
synCancel	107
synChangeMetadata	108

synCopy	109
synCopy	110
synCopy	111
synCopy	112
synCreate	114
synDelete	115
synDeleteColumn	117
synDeleteColumn	118
synDeleteColumn	119
synDeleteColumn	120
synDeleteColumn	121
synDeleteInvitation	122
synDeletePermissions	123
synDeletePermissions	125
synDeletePermissions	128
synDeletePermissions	131
synDeletePermissions	134
synDeletePermissions	136
synDeletePermissions	139
synDeletePermissions	142
synDeletePermissions	145
synDeletePermissions	147
synDeleteProjectSetting	150
synDeleteProjectSetting	151
synDeleteRows	152
synDisassociateActivityFromEntity	154
synDownloadListAdd	155
synDownloadListClear	156
synDownloadListFiles	156
synDownloadListManifest	157
synDownloadListRemove	158
synFindEntityId	159
synFlattenFileList	160
synFlattenFileList	161
synGenerateSyncManifest	162
synGenerateSyncManifest	163
synGet	164
synGetAcl	168
synGetAcl	170
synGetAcl	171
synGetAcl	172
synGetAcl	173
synGetAcl	174
synGetAcl	176
synGetAcl	177
synGetAcl	178
synGetAcl	179
synGetAcl	180

synGetAcl	181
synGetAllEvaluations	182
synGetAllSubmissionStatuses	184
synGetAttachment	185
synGetAttachmentHandles	186
synGetAttachmentPreview	186
synGetAvailableEvaluations	187
synGetChatHistory	189
synGetEvaluationsByProject	189
synGetEvaluationSubmissionBundles	191
synGetEvaluationSubmissions	192
synGetFromId	193
synGetFromId	194
synGetFromId	194
synGetFromName	195
synGetFromParent	196
synGetFromPath	197
synGetFromUsername	198
synGetInvalidValidation	199
synGetInvalidValidation	200
synGetJsonSchemas	202
synGetMarkdownFile	203
synGetOpenInvitations	204
synGetPermissions	205
synGetPermissions	206
synGetPermissions	206
synGetPermissions	207
synGetPermissions	208
synGetPermissions	209
synGetPermissions	210
synGetPermissions	210
synGetPermissions	211
synGetPermissions	212
synGetPermissions	213
synGetProjectSetting	213
synGetProjectSetting	214
synGetSchema	215
synGetSchema	217
synGetSchema	219
synGetSchema	220
synGetSchema	222
synGetSchemaDerivedKeys	223
synGetSchemaDerivedKeys	225
synGetSchemaDerivedKeys	227
synGetSchemaDerivedKeys	229
synGetSchemaDerivedKeys	231
synGetSchemaValidationStatistics	232
synGetSchemaValidationStatistics	234

synGetSession	236
synGetStsStorageToken	237
synGetStsStorageToken	238
synGetSubmissionCount	240
synGetTeamMembers	240
synGetUserMembershipStatus	241
synGetUserSubmissionBundles	242
synGetUserSubmissions	243
synIndexFilesForMigration	244
synIndexFilesForMigration	245
synInviteToTeam	246
synIsCertified	247
synIsSynapseId	248
synList	249
synListAcl	250
synListAcl	252
synListAcl	254
synListAcl	257
synListAcl	259
synListAcl	262
synListAcl	265
synListAcl	267
synListAcl	270
synListAcl	272
synLogin	274
synLogout	276
synMapDirectoryToAllContainedFiles	276
synMapDirectoryToAllContainedFiles	277
synMd5Query	279
synMigrateIndexedFiles	280
synMigrateIndexedFiles	282
synOnweb	283
synPrintEntity	284
synPrompt	285
synPrompt	287
synQuery	288
synQuery	290
synQuery	291
synQuery	292
synQuery	294
synQuery	295
synQuery	297
synQueryPartMask	298
synQueryPartMask	299
synQueryPartMask	301
synQueryPartMask	302
synQueryPartMask	303
synQueryPartMask	304

synQueryPartMask	306
synReformatAttachmentFileName	307
synRegister	308
synRemoveItem	309
synRemoveItem	310
synReorderColumn	311
synReorderColumn	312
synReorderColumn	313
synReorderColumn	314
synReorderColumn	315
synRestDelete	316
synRestGet	317
synRestore	318
synRestPost	319
synRestPut	320
synSendMessage	321
synSetEndpoints	322
synSetPermissions	323
synSetPermissions	325
synSetPermissions	326
synSetPermissions	327
synSetPermissions	328
synSetPermissions	330
synSetPermissions	331
synSetPermissions	332
synSetPermissions	333
synSetPermissions	335
synSetProperties	336
synSetStorageLocation	337
synSetStorageLocation	338
synSnapshot	339
synSnapshot	340
synSnapshot	342
synSnapshot	343
synSnapshot	345
synStart	346
synStartSession	347
synStore	348
synStoreRows	352
synSyncFromSynapse	360
synSyncFromSynapse	365
synSyncToSynapse	369
synSyncToSynapse	370
synUnbindSchema	371
synUnbindSchema	372
synUnbindSchema	374
synUnbindSchema	376
synUnbindSchema	378

synUnzipGzippedFile	379
synUpdate	380
synUpdateAcl	381
synUpdateAcl	383
synUpdateRows	384
synUpdateRows	386
synUpdateRows	388
synUpsertRows	390
synValidateSchema	395
synValidateSchema	397
synValidateSchema	399
synValidateSchema	401
synValidateSchema	403
synWalk	405
synWalk	407
Table	410
TableOptions	420
Team	422
UsedEntity	424
UsedURL	425
UserPreference	426
UserProfile	427
VirtualTable	429
WikiHeader	432
WikiHistorySnapshot	433
WikiOrderHint	434
WikiPage	436
Index	439

Activity	Activity
----------	----------

Description

An activity is a Synapse object that helps to keep track of what objects were used in an analysis step, as well as what objects were generated. Thus, all relationships between Synapse objects and an activity are governed by dependencies. That is, an activity needs to know what it used, and outputs need to know what activity they were generatedBy.

Usage

Activity(id=NULL, name=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_

Arguments

<code>id</code>	(Optional[str]) The unique immutable ID for this activity.
<code>name</code>	(Optional[str]) A name for this Activity.
<code>description</code>	(Optional[str]) A description for this Activity.
<code>etag</code>	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
<code>created_on</code>	(Optional[str]) The date this object was created.
<code>modified_on</code>	(Optional[str]) The date this object was last modified.
<code>created_by</code>	(Optional[str]) The user that created this object.
<code>modified_by</code>	(Optional[str]) The user that last modified this object.
<code>used</code>	(list[Union[UsedEntity, UsedURL]]) The entities or URLs used by this Activity. Only UsedEntity and UsedURL objects are accepted. Other values, such as a plain Synapse ID string (for example "syn123"), are dropped with no error.
<code>executed</code>	(list[Union[UsedEntity, UsedURL]]) The entities or URLs executed by this Activity. Only UsedEntity and UsedURL objects are accepted. Other values, such as a plain Synapse ID string (for example "syn123"), are dropped with no error.

Format

An S3-style class object.

Value

NULL

Methods

- `Activity(id=NULL, name=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, used=list(), executed=list())`: Constructor for [Activity](#)
- `synDisassociateActivityFromEntity(cls, parent, synapse_client=NULL)`: Disassociate the Activity from the parent entity. This is the first step in deleting the Activity. If you have other entities that are associated with this Activity you must disassociate them by calling this method on them as well.
- `synGetFromParent(cls, parent, parent_version_number=NULL, synapse_client=NULL)`: Get the Activity from Synapse based on the parent entity.

Examples

```
## Not run:
## Example 1: Store an Activity on a File
# Record the provenance of a file: what it used, and what was executed to
# create it. Storing the Activity with a parent associates it with that entity.
library(synapser)
```



```

synLogin()

file = synGet("syn123", file_options = FileOptions(download_file = FALSE))

activity = Activity(
  name = "Normalization",
  description = "Normalized the raw counts",
  used = list(
    UsedEntity(target_id = "syn456", target_version_number = 1L),
    UsedURL(name = "Reference genome", url = "https://www.ncbi.nlm.nih.gov/grc/human")
  ),
  executed = list(
    UsedURL(name = "Analysis code", url = "https://github.com/my-org/my-analysis")
  )
)
activity = activity |> synStore(parent = file)
print(activity$id)

## Example 2: Delete an Activity
# Disassociate the Activity from its parent entity. The Activity itself is
# deleted once it has been disassociated from every entity it is attached to;
# until then you'll receive an error for each remaining entity.
library(synapser)
synLogin()

Activity() |> synDelete(parent = "syn123") # parent is the entity that the Activity is associated with

## End(Not run)

```

ActivityOptions

ActivityOptions

Description

Configuration options for entities that support activity/provenance tracking.

This class controls whether activity information (provenance data) should be included when retrieving entities. Activity information tracks the computational steps, data sources, and relationships that led to the creation of an entity.

Usage

```
ActivityOptions(include_activity=FALSE)
```

Arguments

`include_activity`

(bool) Whether to include activity/provenance information when retrieving the entity. If TRUE, the returned entity will have its activity attribute populated with provenance data (if available). If FALSE, the activity attribute will be NULL. Including activity may result in additional API calls and slower retrieval times. Default is FALSE.

Format

An S3-style class object.

Value

NULL

Methods

- `ActivityOptions(include_activity=FALSE)`: Constructor for [ActivityOptions](#)

Note

Activity information is only available for entities that support provenance tracking (File, Table, Dataset, etc...). For other entity types, this option is ignored.

Examples

```
## Not run:
# Configure activity inclusion:
library(synapser)

# Include activity information
with_activity = ActivityOptions(include_activity = TRUE)

# Skip activity information (faster retrieval)
without_activity = ActivityOptions(include_activity = FALSE)

## End(Not run)
```

Agent	<i>Agent</i>
-------	--------------

Description

Represents a [Synapse Agent Registration](#)

Usage

```
Agent(cloud_agent_id=NULL, cloud_alias_id=NULL, registration_id=NULL, registered_on=NULL, type=NULL,
```

Arguments

- `cloud_agent_id` (Optional[str]) The unique ID of the agent in the cloud provider.
- `cloud_alias_id` (Optional[str]) The alias ID of the agent in the cloud provider. Defaults to 'TSTALIASID' in the Synapse API.
- `registration_id` (Optional[int]) The ID number of the agent assigned by Synapse.

registered_on	(Optional[datetime]) The date the agent was registered.
type	(Optional[AgentType]) The type of agent. One of either BASELINE or CUSTOM.
sessions	(dict[str, AgentSession]) A named list of AgentSession objects, keyed by session ID.
current_session	(Optional[AgentSession]) The current session. Prompts will be sent to this session by default.

Format

An S3-style class object.

Value

NULL

Methods

- `Agent(cloud_agent_id=NULL, cloud_alias_id=NULL, registration_id=NULL, registered_on=NULL, type=NULL, sessions=list(), current_session=NULL)`: Constructor for [Agent](#)
- `synGetChatHistory(instance)`: Gets the chat history for the current session.
- `synGetSession(instance, session_id, synapse_client=NULL)`: Gets an existing agent session. Adds the session to the Agent's sessions named list and sets it as the current session.
- `synPrompt(instance, prompt, enable_trace=FALSE, print_response=FALSE, session=NULL, newer_than=NULL, timeout=120L, synapse_client=NULL)`: Sends a prompt to the agent for the current session. If no session is currently active, a new session will be started.
- `synRegister(instance, synapse_client=NULL)`: Registers an agent with the Synapse API. If agent already exists, it will be retrieved.
- `synStartSession(instance, access_level="PUBLICLY_ACCESSIBLE", synapse_client=NULL)`: Starts an agent session. Adds the session to the Agent's sessions named list and sets it as the current session.

Examples

```
## Not run:
## Example 1: Chat with the baseline Synapse Agent
# You can chat with the same agent which is available in the Synapse UI
# at https://www.synapse.org/Chat:default. By default, this "baseline" agent
# is used when a registration ID is not provided. In the background,
# the Agent class will start a session and set that new session as the
# current session if one is not already set.
library(synapser)
synLogin()

my_agent = Agent()
my_agent |> synPrompt(
  prompt = "Can you tell me about the AD Knowledge Portal dataset?",
```

```

        enable_trace = TRUE,
        print_response = TRUE
    )

## Example 2: Register and chat with a custom agent
# Only available for internal users (Sage Bionetworks employees).
#
# Alternatively, you can register a custom agent and chat with it provided
# you have already created it.
library(synapser)
synLogin()

my_agent = Agent(cloud_agent_id = "foo") |> synRegister()
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## Example 3: Prompt an existing custom agent
# If you have already registered a custom agent, you can chat with it by
# constructing an Agent with its registration ID and sending it a prompt.
library(synapser)
synLogin()

my_agent = Agent(registration_id = "foo")
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## Advanced Example: Start and prompt multiple sessions
# Here, we connect to a custom agent and start one session with the prompt "Hello".
# In the background, this first session is being set as the current session
# and future prompts will be sent to this session by default. If we want to send a
# prompt to a different session, we can do so by starting it and calling synPrompt again,
# but with our new session as an argument. We now have two sessions, both stored in the
# my_agent$sessions named list. After the second prompt, my_second_session is now
# the current session.
library(synapser)
synLogin()

my_agent = Agent(registration_id = "foo")

my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

my_second_session = my_agent |> synStartSession()
my_agent |> synPrompt(

```

```
        prompt = "Hello again",
        enable_trace = TRUE,
        print_response = TRUE,
        session = my_second_session
    )

## End(Not run)
```

AgentPrompt	<i>AgentPrompt</i>
-------------	--------------------

Description

Represents a prompt, response, and metadata within an AgentSession.

Usage

```
AgentPrompt(concrete_type="org.sagebionetworks.repo.model.agent.AgentChatRequest", id=NULL, session_
```

Arguments

concrete_type	(str) Indicates the type of request this represents to the Synapse REST API. Should not typically need to be changed from its default value.
id	(Optional[str]) The unique ID of the agent prompt.
session_id	(Optional[str]) The ID of the session that the prompt is associated with.
prompt	(Optional[str]) The prompt to send to the agent.
response	(Optional[str]) The response from the agent.
enable_trace	(Optional[bool]) Whether tracing is enabled for the prompt.
trace	(Optional[str]) The trace of the agent session.

Format

An S3-style class object.

Value

NULL

Methods

- AgentPrompt(concrete_type="org.sagebionetworks.repo.model.agent.AgentChatRequest", id=NULL, session_id=NULL, prompt=NULL, response=NULL, enable_trace=FALSE, trace=NULL): Constructor for [AgentPrompt](#)

Examples

```
## Not run:
## Inspecting the prompts in a session
# AgentPrompt objects are created for you when you prompt an agent. They make up
# the chat history of an AgentSession.
library(synapser)
synLogin()

my_agent = Agent()
my_agent |> synPrompt(prompt = "Hello", print_response = FALSE)

for (agent_prompt in my_agent |> synGetChatHistory()) {
  print(agent_prompt$prompt)
  print(agent_prompt$response)
}

## End(Not run)
```

AgentSession	<i>AgentSession</i>
--------------	---------------------

Description

Represents a **Synapse Agent Session**

Usage

```
AgentSession(id=NULL, access_level="PUBLICLY_ACCESSIBLE", started_on=NULL, started_by=NULL, modified_
```

Arguments

- id (Optional[str]) The unique ID of the agent session. Can only be used by the user that created it.
- access_level (Optional[AgentSessionAccessLevel]) The access level of the agent session. One of PUBLICLY_ACCESSIBLE, READ_YOUR_PRIVATE_DATA, or WRITE_YOUR_PRIVATE_DATA.
- started_on (Optional[datetime]) The date the agent session was started.
- started_by (Optional[int]) The ID of the user who started the agent session.
- modified_on (Optional[datetime]) The date the agent session was last modified.
- agent_registration_id (Optional[int]) The registration ID of the agent that will be used for this session.
- etag (Optional[str]) The etag of the agent session.
- chat_history (List[AgentPrompt]) A list of AgentPrompt objects.

Format

An S3-style class object.

Value

NULL

Methods

- `AgentSession(id=NULL, access_level="PUBLICLY_ACCESSIBLE", started_on=NULL, started_by=NULL, modified_on=NULL, agent_registration_id=NULL, etag=NULL, chat_history=list())`: Constructor for [AgentSession](#)
- `synPrompt(instance, prompt, enable_trace=FALSE, print_response=FALSE, newer_than=NULL, timeout=120L, synapse_client=NULL)`: Sends a prompt to the agent and adds the response to the AgentSession's chat history. A session must be started before sending a prompt.
- `synStart(instance, synapse_client=NULL)`: Starts an agent session.
- `synUpdate(instance, synapse_client=NULL)`: Updates an agent session. Only updates to the access level are currently supported.

Note

It is recommended to use the [Agent](#) class to conduct chat sessions, but you are free to use `AgentSession` directly if you wish.

Examples

```
## Not run:
## Example 1: Start a session and send a prompt.
# Start a session with a custom agent by providing the agent's registration ID
# and calling synStart(). Then, send a prompt to the agent.
library(synapser)
synLogin()

my_session = AgentSession(agent_registration_id = "foo") |> synStart()
my_session |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## Example 2: Prompt an existing session.
# If you already have an existing session, you can send it a prompt by
# constructing an AgentSession with its session ID.
library(synapser)
synLogin()

my_session = AgentSession(id = "foo")
my_session |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## Example 3: Update the access level of an existing session.
```

```
# Construct an AgentSession with its session ID, set the access level you
# want, and call synUpdate().
library(synapser)
synLogin()

my_session = AgentSession(id = "foo")
my_session$access_level = "READ_YOUR_PRIVATE_DATA"
my_session = my_session |> synUpdate()

## End(Not run)
```

Column	<i>Column</i>
--------	---------------

Description

A column model contains the metadata of a single column of a table or view.

Usage

```
Column(id=NULL, name=NULL, column_type=NULL, facet_type=NULL, default_value=NULL, maximum_size=NULL, ...)
```

Arguments

id	(Optional[str]) The immutable ID issued to new columns.
name	(Optional[str]) The display name of the column.
column_type	(Optional[ColumnType]) The column type determines the type of data that can be stored in a column. Switching between types (using a transaction with TableUpdateTransaction in the "changes" list) is generally allowed except for switching to "_LIST" suffixed types. In such cases, a new column must be created and data must be copied over manually.
facet_type	(Optional[FacetType]) Set to one of the enumerated values to indicate a column should be treated as a facet.
default_value	(Optional[str]) The default value for this column. Columns of type ENTITYID, FILEHANDLEID, USERID, and LARGETEXT are not allowed to have default values.
maximum_size	(Optional[int]) A parameter for column types with a maximum size. For example, columns of type "STRING" have a default maximum size of 50 characters, but can be set to a maximum size of 1 to 1000 characters. For a column type of STRING_LIST, this limits the size of individual string elements in the list.
maximum_list_length	(Optional[int]) Required if using a column type with a "_LIST" suffix. Describes the maximum number of values that will appear in that list. Value range 1-100 inclusive. Default 100.
enum_values	(Optional[list[str]]) Columns of type STRING can be constrained to an enumeration of values set on this list. The maximum number of entries for an enum is 100.

json_sub_columns

(Optional[list[JsonSubColumn]]) For a column of type JSON that represents the combination of multiple sub-columns, this property is used to define each sub-column.

Format

An S3-style class object.

Value

NULL

Methods

- Column(id=NULL, name=NULL, column_type=NULL, facet_type=NULL, default_value=NULL, maximum_size=NULL, maximum_list_length=NULL, enum_values=NULL, json_sub_columns=NULL): Constructor for [Column](#)
- [synList](#)(prefix=NULL, limit=100L, offset=0L, synapse_client=NULL): List columns with optional prefix filtering.

Examples

```
## Not run:
## Creating a table with columns
library(synpser)
synLogin()

columns = list(
  Column(name = "my_string_column", column_type = "STRING", maximum_size = 100L),
  Column(name = "my_integer_column", column_type = "INTEGER"),
  Column(name = "my_status_column", column_type = "STRING", enum_values = list("pending", "done")),
  Column(name = "my_list_column", column_type = "STRING_LIST", maximum_list_length = 10L)
)
table = Table(name = "My Table", parent_id = "syn123", columns = columns) |> synStore()

## Listing existing column models
library(synpser)
synLogin()

columns = synList(prefix = "my_", limit = 10L)

## End(Not run)
```

Dataset	<i>Dataset</i>
---------	----------------

Description

A Dataset object represents the metadata of a Synapse Dataset. <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/table/Dataset.html>

Usage

```
Dataset(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type="dataset")
```

Arguments

id	(Optional[character]) The unique immutable ID for this dataset. A new ID will be generated for new Datasets. Once issued, this ID is guaranteed to never change or be re-issued
name	(Optional[character]) The name of this dataset. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses
parent_id	(Optional[character]) The ID of the Entity that is the parent of this dataset.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance.
version_number	(Optional[integer]) The version number issued to this version on the object.
view_entity_type	(character) The API model string for the type of view. This is used to determine the default columns that are added to the table. Defaults to "dataset".
view_type_mask	(integer) The Bit Mask representing Dataset type. As defined in the Synapse REST API: https://rest-docs.synapse.org/rest/GET/column/tableview/defaults.html . Defaults to 128.
include_default_columns	<p>(Optional[logical]) When creating a dataset or view, specifies if default columns should be included. Default columns are columns that are automatically added to the dataset or view. These columns are managed by Synapse and cannot be modified. If you attempt to create a column with the same name as a default column, you will receive a warning when you store the dataset.</p> <p>include_default_columns is only used if this is the first time that the view is being stored. If you are updating an existing view this attribute will be ignored.</p> <p>The column you are overriding will not behave the same as a default column. For example, suppose you create a column called id on a Dataset. When using a default column, the id stores the Synapse ID of each of the entities included in</p>

the scope of the view. If you override the `id` column with a new column, the `id` column will no longer store the Synapse ID of the entities in the view. Instead, it will store the values you provide when you store the dataset. It will be stored as an annotation on the entity for the row you are modifying.

<code>description</code>	(Optional[character]) The description of the dataset. Must be 1000 characters or less.
<code>etag</code>	(Optional[character]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
<code>created_on</code>	(Optional[character]) The date this dataset was created.
<code>modified_on</code>	(Optional[character]) The date this dataset was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format
<code>created_by</code>	(Optional[character]) The ID of the user that created this dataset.
<code>modified_by</code>	(Optional[character]) The ID of the user that last modified this dataset.
<code>version_label</code>	(Optional[character]) The version label for this dataset.
<code>version_comment</code>	(Optional[character]) The version comment for this dataset.
<code>is_latest_version</code>	(Optional[logical]) If this is the latest version of the object.
<code>is_search_enabled</code>	(Optional[logical]) When creating or updating a dataset or view specifies if full text search should be enabled. Note that enabling full text search might slow down the indexing of the dataset or view.
<code>items</code>	(Optional[list[EntityRef]]) The flat list of file entity references that define this dataset. This is effectively a list of the rows that are in/will be in the dataset after it is stored. The only way to add or remove rows is to add or remove items from this list.
<code>size</code>	(Optional[integer]) The cumulative size, in bytes, of all items (files) in the dataset. This is only correct after the dataset has been stored or newly read from Synapse.
<code>checksum</code>	(Optional[character]) The checksum is computed over a sorted concatenation of the checksums of all items in the dataset. This is only correct after the dataset has been stored or newly read from Synapse.
<code>count</code>	(Optional[integer]) The number of items/files in the dataset. This is only correct after the dataset has been stored or newly read from Synapse.
<code>columns</code>	(Optional; a list of Column objects or a named list of Column objects) The columns of this dataset. This is an ordered named list where the key is the name of the column and the value is the Column object. When creating a new instance of a Dataset object you may pass any of the following types as the <code>columns</code> argument: <ul style="list-style-type: none"> - A list of Column objects - A named list where the key is the name of the column and the value is the Column object

The order of the columns will be the order they are stored in Synapse. If you need to reorder the columns the recommended approach is to use the `synReorderColumn()` function. Additionally, you may add, and delete columns using the `synAddColumn()`, and `synDeleteColumn()` functions on your dataset class instance.

You may modify the attributes of the Column object to change the column type, name, or other attributes. For example, suppose you'd like to change a column from a INTEGER to a DOUBLE. You can do so by changing the column type attribute of the Column object. The next time you store the dataset the column will be updated in Synapse with the new type.

```
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet()
dataset$columns[["my_column"]][column_type = "DOUBLE"
dataset = dataset |> synStore()
```

Note that the keys in this named list should match the column names as they are in Synapse. However, know that the name attribute of the Column object is used for all interactions with the Synapse API. The named list key is purely for the usage of this interface. For example, if you wish to rename a column you may do so by changing the name attribute of the Column object. The key in the named list does not need to be changed. The next time you store the dataset the column will be updated in Synapse with the new name and the key in the named list will be updated.

annotations (Optional[named list]) Additional metadata associated with the dataset. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list.

Format

An S3-style class object.

Value

NULL

Methods

- `Dataset(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type="dataset", view_type_mask=128L, include_default_columns=TRUE, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, is_search_enabled=NULL, items=list(), size=NULL, checksum=NULL, count=NULL, columns=list(), annotations=list())`: Constructor for [Dataset](#)

- `synAddColumn(instance, column, index=NULL)`: Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this Dataset instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the Dataset instance. `$columns` is a named list where the key is the name of the column and the value is the Column object.

- `synAddItem(instance, item, synapse_client=NULL)`: Adds an item in the form of an `EntityRef` to the dataset. For Folders, children are added recursively. Effect is not seen until the dataset is stored.
- `synDeleteColumn(instance, name)`: Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this Dataset instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

- `synRemoveItem(instance, item, synapse_client=NULL)`: Removes an item from the dataset. For Folders, all children of the folder are removed recursively. Effect is not seen until the dataset is stored.
- `synReorderColumn(instance, name, index)`: Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this Dataset instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_timeout=120L, synapse_client=NULL)`: Creates a snapshot of the dataset. A snapshot is a saved, read-only version of the dataset at the time it was created. Dataset snapshots are created using the asynchronous job API.
- `synUpdateRows(instance, values, primary_keys, dry_run=FALSE, rows_per_query=50000L, update_size_bytes=1992294.4, insert_size_bytes=943718400L, job_timeout=600L, date_columns=NULL, date_format=NULL, wait_for_eventually_consistent_view=FALSE, wait_for_eventually_consistent_view_timeout=600L, synapse_client=NULL)`: Update the values of rows in the dataset. This method can only be used to update values in custom columns. Default columns cannot be updated, but may be used as primary keys.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an error will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

Examples

```
## Not run:
## Example 1: Create a new dataset from a list of EntityRefs.
# Dataset items consist of references to Synapse Files using an Entity Reference.
# If you are adding items to a Dataset directly, you must provide them in the form of
# an EntityRef class instance.
library(synapser)
synLogin()

my_entity_refs = list(EntityRef(id = "syn1234"), EntityRef(id = "syn1235"), EntityRef(id = "syn1236"))
my_dataset = Dataset(parent_id = "syn987", name = "my-new-dataset", items = my_entity_refs)
my_dataset |> synStore()

## Example 2: Add entities to an existing dataset.
# Using synAddItem(), you can add Synapse entities that are Files, Folders, or EntityRefs that point to a Synapse
# If the entity is a Folder (or an EntityRef that points to a folder), all of the child Files
# within the Folder will be added to the Dataset recursively.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()

# Add a file to the dataset
my_dataset |> synAddItem(File(id = "syn1235"))

# Add a folder to the dataset
# All child files are recursively added to the dataset
my_dataset |> synAddItem(Folder(id = "syn1236"))

# Add an entity reference to the dataset
my_dataset |> synAddItem(EntityRef(id = "syn1237", version = 1L))

my_dataset |> synStore()

## Example 3: Remove entities from a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
```

```

# Remove a file from the dataset
my_dataset |> synRemoveItem(File(id = "syn1235"))

# Remove a folder from the dataset
# All child files are recursively removed from the dataset
my_dataset |> synRemoveItem(Folder(id = "syn1236"))

# Remove an entity reference from the dataset
my_dataset |> synRemoveItem(EntityRef(id = "syn1237", version = 1L))

my_dataset |> synStore()

## Example 4: Query data from a dataset.
library(synapser)
synLogin()

row = synQuery(query = "SELECT * FROM syn1234 WHERE id = 'syn1235'")
print(row)

## Example 5: Add a custom column to a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synAddColumn(Column(name = "my_annotation", column_type = "STRING"))
my_dataset |> synStore()

## Example 6: Update custom column values in a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
# my_annotation must already exist in the dataset as a custom column
modified_data = data.frame(
  id = c("syn1234"),
  my_annotation = c("good data")
)
my_dataset |> synUpdateRows(values = modified_data, primary_keys = list("id"), dry_run = FALSE)

## Example 7: Save a snapshot of a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synSnapshot(comment = "My first snapshot", label = "My first snapshot")

## Example 8: Deleting a dataset.
library(synapser)
synLogin()

Dataset(id = "syn4567") |> synDelete()

```



```
## End(Not run)
```

DatasetCollection	<i>DatasetCollection</i>
-------------------	--------------------------

Description

A DatasetCollection object represents the metadata of a Synapse Dataset Collection. <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/table/DatasetCollection.html>

A Dataset Collection is a type of view defined by a flat list of Datasets.

Usage

```
DatasetCollection(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type=NULL)
```

Arguments

id	(Optional[character]) The unique immutable ID for this dataset collection. A new ID will be generated for new DatasetCollections. Once issued, this ID is guaranteed to never change or be re-issued
name	(Optional[character]) The name of this dataset collection. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses
parent_id	(Optional[character]) The ID of the Entity that is the parent of this dataset collection.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance.
version_number	(Optional[integer]) The version number issued to this version on the object.
view_entity_type	(character) The API model string for the type of view. This is used to determine the default columns that are added to the table. Defaults to "datasetcollection".
view_type_mask	(integer) The Bit Mask representing DatasetCollection type. As defined in the Synapse REST API: https://rest-docs.synapse.org/rest/GET/column/tableview/defaults.html . Defaults to 256.
include_default_columns	(Optional[logical]) When creating a dataset collection or view, specifies if default columns should be included. Default columns are columns that are automatically added to the dataset collection or view. These columns are managed by Synapse and cannot be modified. If you attempt to create a column with the same name as a default column, you will receive a warning when you store the dataset collection.

include_default_columns is only used if this is the first time that the view

is being stored. If you are updating an existing view this attribute will be ignored.

The column you are overriding will not behave the same as a default column. For example, suppose you create a column called `id` on a `DatasetCollection`. When using a default column, the `id` stores the Synapse ID of each of the entities included in the scope of the view. If you override the `id` column with a new column, the `id` column will no longer store the Synapse ID of the entities in the view. Instead, it will store the values you provide when you store the dataset collection. It will be stored as an annotation on the entity for the row you are modifying.

<code>description</code>	(Optional[character]) The description of the dataset collection. Must be 1000 characters or less.
<code>etag</code>	(Optional[character]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
<code>created_on</code>	(Optional[character]) The date this dataset collection was created.
<code>modified_on</code>	(Optional[character]) The date this dataset collection was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format
<code>created_by</code>	(Optional[character]) The ID of the user that created this dataset collection.
<code>modified_by</code>	(Optional[character]) The ID of the user that last modified this dataset collection.
<code>version_label</code>	(Optional[character]) The version label for this dataset collection.
<code>version_comment</code>	(Optional[character]) The version comment for this dataset collection.
<code>is_latest_version</code>	(Optional[logical]) If this is the latest version of the object.
<code>is_search_enabled</code>	(Optional[logical]) When creating or updating a dataset collection or view specifies if full text search should be enabled. Note that enabling full text search might slow down the indexing of the dataset collection or view.
<code>items</code>	(Optional[list[EntityRef]]) The flat list of datasets that define this collection. This is effectively a list of the rows that are in/will be in the collection after it is stored. The only way to add or remove rows is to add or remove items from this list.
<code>columns</code>	(Optional[Union[list[Column], OrderedDict[str, Column], dict[str, Column]]]) The columns of this dataset collection. This is an ordered named list where the key is the name of the column and the value is the Column object. When creating a new instance of a <code>DatasetCollection</code> object you may pass any of the following types as the <code>columns</code> argument: <ul style="list-style-type: none"> - A list of Column objects - A named list where the key is the name of the column and the value is the Column object

The order of the columns will be the order they are stored in Synapse. If you need to reorder the columns the recommended approach is to use the `synReorderColumn()` function. Additionally, you may add, and delete columns using the `synAddColumn()`, and `synDeleteColumn()` functions on your dataset collection class instance.

You may modify the attributes of the Column object to change the column type, name, or other attributes. For example, suppose you'd like to change a column from a INTEGER to a DOUBLE. You can do so by changing the column type attribute of the Column object. The next time you store the dataset collection the column will be updated in Synapse with the new type.

```
library(synapser)
synLogin()
```

```
collection = DatasetCollection(id = "syn1234") |> synGet()
collection$columns[["my_column"]]$column_type = "DOUBLE"
collection = collection |> synStore()
```

Note that the keys in this named list should match the column names as they are in Synapse. However, know that the name attribute of the Column object is used for all interactions with the Synapse API. The named list key is purely for the usage of this interface. For example, if you wish to rename a column you may do so by changing the name attribute of the Column object. The key in the named list does not need to be changed. The next time you store the dataset collection the column will be updated in Synapse with the new name and the key in the named list will be updated.

annotations (Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the dataset collection. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list.

Format

An S3-style class object.

Value

NULL

Methods

- `DatasetCollection(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type="datasetcollection", view_type_mask=256L, include_default_columns=TRUE, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, is_search_enabled=NULL, items=list(), columns=list(), annotations=list())`: Constructor for [DatasetCollection](#)

- `synAddColumn(instance, column, index=NULL)`: Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this DatasetCollection instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the DatasetCollection instance. `$columns` is a named list where the key is the name of the column and the value is the Column object.

- `synAddItem(instance, item)`: Adds a dataset to the dataset collection. Effect is not seen until the dataset collection is stored.
- `synDeleteColumn(instance, name)`: Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this DatasetCollection instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

- `synRemoveItem(instance, item)`: Removes an entity from the dataset collection. Must be a Dataset or EntityRef. Effect is not seen until the dataset collection is stored. Unless the version is specified, all entities with the same ID will be removed.
- `synReorderColumn(instance, name, index)`: Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this DatasetCollection instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_timeout=120L, synapse_client=NULL)`: Creates a snapshot of the dataset collection. A snapshot is a saved, read-only version of the dataset collection at the time it was created. Dataset collection snapshots are created using the asynchronous job API.
- `synUpdateRows(instance, values, primary_keys, dry_run=FALSE, rows_per_query=50000L, update_size_bytes=1992294.4, insert_size_bytes=943718400L, job_timeout=600L, date_columns=NULL, date_format=NULL, wait_for_eventually_consistent_view=FALSE, wait_for_eventually_consistent_view_timeout=600L, synapse_client=NULL)`: Update the values of rows in the dataset collection. This method can only be used to update values in custom columns. Default columns cannot be updated, but may be used as primary keys.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an error will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

Examples

```
## Not run:
## Example 1: Create a new Dataset Collection from a list of Datasets.
library(synapser)
synLogin()

my_datasets = list(Dataset(id = "syn1234"), Dataset(id = "syn1235"), Dataset(id = "syn1236"))
my_collection = DatasetCollection(parent_id = "syn987", name = "my-new-collection", items = my_datasets)
my_collection |> synStore()

## Example 2: Add Datasets to an existing Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()

# Add a dataset to the collection
my_collection |> synAddItem(Dataset(id = "syn1235"))
my_collection |> synStore()

## Example 3: Remove Datasets from a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()

# Remove a dataset from the collection
my_collection |> synRemoveItem(Dataset(id = "syn1235"))
my_collection |> synStore()

## Example 4: Query data from a Dataset Collection.
library(synapser)
synLogin()

row = synQuery(query = "SELECT * FROM syn1234 WHERE id = 'syn1235'")
print(row)

## Example 5: Add a custom column to a Dataset Collection.
library(synapser)
```

```

synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synAddColumn(Column(name = "my_annotation", column_type = "STRING"))
my_collection |> synStore()

## Example 6: Update custom column values in a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
# my_annotation must already exist in the dataset collection as a custom column
modified_data = data.frame(
  id = c("syn1234"),
  my_annotation = c("good data")
)
my_collection |> synUpdateRows(values = modified_data, primary_keys = list("id"), dry_run = FALSE)

## Example 7: Save a snapshot of a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synSnapshot(comment = "My first snapshot", label = "My first snapshot")

## Example 8: Deleting a Dataset Collection.
library(synapser)
synLogin()

DatasetCollection(id = "syn4567") |> synDelete()

## End(Not run)

```

DownloadListItem

*DownloadListItem***Description**

A single item for a user's download list.

<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/download/DownloadListItem.html>

Usage

```
DownloadListItem(file_entity_id, version_number=NULL)
```

Arguments

`file_entity_id` (character) Synapse ID of the file entity (e.g. "syn123").

`version_number` (Optional[integer]) Version of the file to target.

Format

An S3-style class object.

Value

NULL

Methods

- DownloadListItem(file_entity_id, version_number=NULL): Constructor for [DownloadListItem](#)

Examples

```
## Not run:
## Using this class
# Identify files (and optionally their versions) on your download list
library(synapser)
synLogin()

items = list(
  DownloadListItem(file_entity_id = "syn123"),
  DownloadListItem(file_entity_id = "syn456", version_number = 2L)
)
# Add files to the cart
count = synDownloadListAdd(items)
print(count)

# Download all files in the cart to a local directory.
items = synDownloadListFiles()
print(items)

# Remove files from the cart
count = synDownloadListRemove(items)
print(count)

## End(Not run)
```

EntityRef	<i>EntityRef</i>
-----------	------------------

Description

Represents a reference to the id and version of an entity to be used in Dataset and DatasetCollection objects.

Usage

EntityRef(id, version)

Arguments

- id (character) The Synapse ID of the entity.
- version (integer) Indicates a specific version of the entity.

Format

An S3-style class object.

Value

NULL

Methods

- EntityRef(id, version): Constructor for EntityRef

Examples

```
## Not run:
## Adding a specific entity version to a Dataset
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn123") |> synGet()
my_dataset |> synAddItem(EntityRef(id = "syn456", version = 1L))
my_dataset = my_dataset |> synStore()

## End(Not run)
```

EntityView	EntityView
------------	------------

Description

A view of Entities within a defined scope. The purpose of an EntityView, also known as a FileView, is to create a SQL-like view of entities within a defined scope. The scope is defined by the scope_ids attribute. The scope_ids attribute is a list of syn container ids that define where to search for rows to include in the view. Entities within the scope are included in the view if they match the criteria defined by the view_type_mask attribute. The view_type_mask attribute is a bit mask representing the types to include in the view. You may set this to a single integer value, or you may combine multiple values using a bitwise OR (bitwOr() in R).

Usage

EntityView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type=N

Arguments

id	(Optional[str]) The unique immutable ID for this entity. A new ID will be generated for new Entities. Once issued, this ID is guaranteed to never change or be re-issued.
name	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses.
parent_id	(Optional[str]) The ID of the Entity that is the parent of this entity.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance. Activity cannot be removed during a store operation by setting it to NULL. You must use: synDisassociateActivityFromEntity() .
version_number	(Optional[int]) The version number issued to this version on the object.
view_entity_type	(Optional[str]) The type of view. This is used to determine the default columns that are added to the view. One of "entityview", "submissionview", "dataset", or "datasetcollection", as defined by ViewEntityType .
view_type_mask	(Optional[int]) Bit mask representing the types to include in the view. You may set this to a single value, or you may combine multiple values using a bitwise OR. When this is returned after storing or reading from Synapse it will be returned as an integer.

The following are the possible types (type=):

- File=0x01
- Project=0x02
- Table=0x04
- Folder=0x08
- View=0x10
- Docker=0x20
- SubmissionView=0x40
- Dataset=0x80
- DatasetCollection=0x100
- MaterializedView=0x200

To include multiple types in the view you will be using the bitwise OR operator to combine the types. For example, if you want to include both Files and Folders in the view you would use the following code:

```
library(synapser)
synLogin()

view = EntityView(
  name = "My EntityView",
  parent_id = "syn1234",
```

```

scope_ids = list("syn1234"),
view_type_mask = bitwOr(0x01L, 0x08L) # File | Folder
) |> synStore()

```

include_default_columns

(Optional[bool]) When creating an EntityView or view, specifies if default columns should be included. Default columns are columns that are automatically added to the EntityView or view. These columns are managed by Synapse and cannot be modified. If you attempt to create a column with the same name as a default column, you will receive a warning when you store the EntityView.

include_default_columns is only used if this is the first time that the view is being stored. If you are updating an existing view this attribute will be ignored. There is currently no R function to add the default columns back to an existing view.

The column you are overriding will not behave the same as a default column. For example, suppose you create a column called `id` on an EntityView. When using a default column, the `id` stores the Synapse ID of each of the entities included in the scope of the view. If you override the `id` column with a new column, the `id` column will no longer store the Synapse ID of the entities in the view. Instead, it will store the values you provide when you store the EntityView. It will be stored as an annotation on the entity for the row you are modifying.

description

(Optional[str]) The description of this entity. Must be 1000 characters or less.

etag

(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.

created_on

(Optional[str]) The date this entity was created.

modified_on

(Optional[str]) The date this entity was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format.

created_by

(Optional[str]) The ID of the user that created this entity.

modified_by

(Optional[str]) The ID of the user that last modified this entity.

version_label

(Optional[str]) The version label for this entity.

version_comment

(Optional[str]) The version comment for this entity.

is_latest_version

(Optional[bool]) If this is the latest version of the object.

columns

(Optional[Union[list[Column], OrderedDict[str, Column], dict[str, Column]]]) The columns of this view. This is an ordered named list where the key is the name of the column and the value is the Column object. When creating a new instance of an EntityView object you may pass any of the following types as the columns argument:

- A list of Column objects
- A named list where the key is the name of the column and the value is the

Column object

The order of the columns will be the order they are stored in Synapse. If you need to reorder the columns the recommended approach is to use the `synReorderColumn()` function. Additionally, you may add, and delete columns using the `synAddColumn()`, and `synDeleteColumn()` functions on your view class instance.

You may modify the attributes of the Column object to change the column type, name, or other attributes. For example, suppose you'd like to change a column from a `INTEGER` to a `DOUBLE`. You can do so by changing the column type attribute of the Column object. The next time you store the view the column will be updated in Synapse with the new type.

```
library(synapser)
synLogin()

view = EntityView(id = "syn1234") |> synGet()
view$columns[["my_column"]]$column_type = "DOUBLE"
view = view |> synStore()
```

Note that the keys in this named list should match the column names as they are in Synapse. However, know that the name attribute of the Column object is used for all interactions with the Synapse API. The named list key is purely for the usage of this interface. For example, if you wish to rename a column you may do so by changing the name attribute of the Column object. The key in the named list does not need to be changed. The next time you store the view the column will be updated in Synapse with the new name and the key in the named list will be updated.

<code>is_search_enabled</code>	(Optional[bool]) When creating or updating an EntityView or view specifies if full text search should be enabled. Note that enabling full text search might slow down the indexing of the EntityView or view.
<code>scope_ids</code>	(Optional[set[str]]) The list of container ids that define the scope of this view. This may be a single container or multiple containers. A container in this context may refer to a Project or Folder which contains zero or more entities. The entities in the container(s) will be included in the view if they match the criteria defined by the <code>view_type_mask</code> attribute.
<code>annotations</code>	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the EntityView. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.

Format

An S3-style class object.

Value

NULL

Methods

- `EntityView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type=NULL, view_type_mask=NULL, include_default_columns=TRUE, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, columns=list(), is_search_enabled=NULL, scope_ids=list(), annotations=list())`: Constructor for [EntityView](#)

- `synAddColumn(instance, column, index=NULL)`: Add column(s) to the EntityView. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this EntityView instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the EntityView instance. `$columns` is a named list where the key is the name of the column and the value is the Column object.

- `synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)`: Bind a JSON schema to the entity.
- `synDeleteColumn(instance, name)`: Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this EntityView instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the [ACL](#) that a user or group has on an Entity.

- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synGetSchema(instance, synapse_client=NULL)`: Get the JSON schema bound to the entity.
- `synGetSchemaDerivedKeys(instance, synapse_client=NULL)`: Retrieve derived JSON schema keys for the entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

- `synReorderColumn(instance, name, index)`: Reorder a column in the EntityView. Note that this does not store the column in Synapse. You must call the `synStore()` function on this EntityView instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the EntityView. If you pass in an index that is out of bounds the column will be added to the end of the list.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_timeout=120L, synapse_client=NULL)`: Creates a snapshot of the EntityView entity. Synapse

handles snapshot creation differently for Table- and View-like entities. View snapshots are created using the asynchronous job API.

Making a snapshot of a view allows you to create an immutable version of the view at the time of the snapshot. This is useful to create checkpoints in time that you may go back and reference, or use in a publication. Snapshots are immutable and cannot be changed. They may only be deleted.

- `synUnbindSchema(instance, synapse_client=NULL)`: Unbind the JSON schema bound to the entity.
- `synUpdateRows(instance, values, primary_keys, dry_run=FALSE, rows_per_query=50000L, update_size_bytes=1992294.4, insert_size_bytes=943718400L, job_timeout=600L, date_columns=NULL, date_format=NULL, wait_for_eventually_consistent_view=FALSE, wait_for_eventually_consistent_view_timeout=600L, synapse_client=NULL)`: This method leverages the logic provided by `synUpsertRows()` to provide an interface for updating rows in a View-like entity. Update functionality will only work for values in custom columns within a View-like entity.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The `primary_keys` columns must be present in the data being updated. An error is raised if a primary key column is missing.
- The values in the `primary_keys` columns cannot be null. A null value cannot be used to match an existing row, so an error is raised if any row has a null value in a primary key column.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an error will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.
- `synValidateSchema(instance, synapse_client=NULL)`: Validate the entity against the bound JSON schema.

Examples

```
## Not run:
## Creating an EntityView of the files in a project
library(synapser)
synLogin()

view = EntityView(
  name = "My File View",
  parent_id = "syn123",
  scope_ids = list("syn123"),
```

```

        view_type_mask = 1L # 1 = files
    ) |> synStore()

    results = synQuery(query = sprintf("SELECT * FROM %s", view$id))
    print(results)

## End(Not run)

```

Evaluation

Evaluation

Description

An Evaluation is the core object of the Evaluation API, used to support collaborative data analysis challenges in Synapse.

An Evaluation object represents an evaluation queue in Synapse: <https://rest-docs.synapse.org/rest/org/sagebionetworks/evaluation/model/Evaluation.html>

Usage

```
Evaluation(id=NULL, etag=NULL, name=NULL, description=NULL, owner_id=NULL, created_on=NULL, content_s
```

Arguments

<code>id</code>	(Optional[str]) The unique immutable ID for this Evaluation.
<code>etag</code>	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. The eTag changes every time an Evaluation is updated; it is used to detect when a client's copy of an Evaluation is out-of-date.
<code>name</code>	(Optional[str]) **Required to store.** The name of this Evaluation.
<code>description</code>	(Optional[str]) **Required to store.** A text description of this Evaluation.
<code>owner_id</code>	(Optional[str]) The ID of the Synapse user who created this Evaluation.
<code>created_on</code>	(Optional[str]) The date on which Evaluation was created.
<code>content_source</code>	(Optional[str]) **Required to store.** The Synapse ID of the Entity to which this Evaluation belongs, e.g. a reference to a Synapse project.
<code>submission_instructions_message</code>	(Optional[str]) **Required to store.** Message to display to users detailing acceptable formatting for Submissions to this Evaluation.
<code>submission_receipt_message</code>	(Optional[str]) **Required to store.** Message to display to users upon successful submission to this Evaluation.

Format

An S3-style class object.

Value

NULL

Methods

- `Evaluation(id=NULL, etag=NULL, name=NULL, description=NULL, owner_id=NULL, created_on=NULL, content_source=NULL, submission_instructions_message=NULL, submission_receipt_message=NULL)`: Constructor for [Evaluation](#)
- `synGetAcl(instance, synapse_client=NULL)`: Get the access control list (ACL) governing this evaluation.
- `synGetAllEvaluations(access_type=NULL, active_only=NULL, evaluation_ids=NULL, offset=NULL, limit=NULL, synapse_client=NULL)`: Get a list of all Evaluations, within a given range.
- `synGetAvailableEvaluations(active_only=NULL, evaluation_ids=NULL, offset=NULL, limit=NULL, synapse_client=NULL)`: Get a list of Evaluations to which the user has SUBMIT permission, within a given range.
- `synGetEvaluationsByProject(project_id, access_type=NULL, active_only=NULL, evaluation_ids=NULL, offset=NULL, limit=NULL, synapse_client=NULL)`: Get Evaluations tied to a project.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the user permissions for this evaluation.
- `synUpdateAcl(instance, principal_id=NULL, access_type=NULL, acl=NULL, synapse_client=NULL)`: Update the access control list (ACL) for this evaluation.

You can either:

1. Provide a `principal_id` and `access_type` list to update permissions for a specific user/team
2. Provide a complete ACL named list to update all permissions at once

To remove a principal from the ACL completely, provide an empty list for `access_type`.

The available access types are:

- "CREATE"
- "SUBMIT"
- "READ_PRIVATE_SUBMISSION"
- "DELETE_SUBMISSION"
- "UPDATE_SUBMISSION"
- "CHANGE_PERMISSIONS"
- "READ"
- "DELETE"
- "UPDATE"

Examples

```
## Not run:
## Example 1: Create a new evaluation in a project
# Create a new evaluation on Synapse by storing an evaluation object with the required fields. If there are any fi
library(synapser)
```

```

synLogin()

evaluation = Evaluation(
  name = "My Challenge Evaluation",
  description = "Evaluation for my data challenge",
  content_source = "syn123456",
  submission_instructions_message = "Submit CSV files only",
  submission_receipt_message = "Thank you for your submission!"
)
created = evaluation |> synStore()

## Example 2: Get an evaluation by ID and by name
library(synapser)
synLogin()

# Get an evaluation by ID
evaluation_by_id = Evaluation(id = "9999999") |> synGet()

# Get an evaluation by name
evaluation_by_name = Evaluation(name = "My Challenge Evaluation") |> synGet()

## Example 3: Update an existing evaluation retrieved from Synapse by ID
# You can use synStore() to create and update evaluations.
library(synapser)
synLogin()

evaluation = synGetAllEvaluations(evaluation_ids = list("9999999"))[[1]]
evaluation$description = "Updated description for my evaluation"
updated = evaluation |> synStore()

## Example 4: Retrieve and update the ACL of an evaluation
library(synapser)
synLogin()

evaluation = synGetAllEvaluations(evaluation_ids = list("9999999"))[[1]]
acl = evaluation |> synGetAcl()

# Let's grant this team READ and SUBMIT permissions
team_id = "123456"
evaluation |> synUpdateAcl(principal_id = team_id, access_type = list("READ", "SUBMIT"))

# Now let's revoke all permissions from this team
team_to_revoke_from = "654321"
evaluation |> synUpdateAcl(principal_id = team_to_revoke_from, access_type = list())

## Example 5: Delete an evaluation
library(synapser)
synLogin()

evaluation = synGetAllEvaluations(evaluation_ids = list("9999999"))[[1]]
evaluation |> synDelete()

## End(Not run)

```

File

*File***Description**

A file within Synapse.

Usage

```
File(id=NULL, name=NULL, path=NULL, description=NULL, parent_id=NULL, version_label=NULL, version_comment=NULL)
```

Arguments

id	(Optional[str]) The unique immutable ID for this file. A new ID will be generated for new Files. Once issued, this ID is guaranteed to never change or be re-issued.
name	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses. If not specified, the name will be derived from the file name.
path	(Optional[str]) The path to the file on disk. Using shorthand ~ will be expanded to the user's home directory. This is used during a get operation to specify where to download the file to. It should be pointing to a directory. This is also used during a store operation to specify the file to upload. It should be pointing to a file.
description	(Optional[str]) The description of this file. Must be 1000 characters or less.
parent_id	(Optional[str]) The ID of the Entity that is the parent of this Entity. Setting this to a new value and storing it will move this File under the new parent.
version_label	(Optional[str]) The version label for this entity. Updates to the entity will increment the version number.
version_comment	(Optional[str]) The version comment for this entity
data_file_handle_id	(Optional[str]) ID of the file associated with this entity. You may define an existing data_file_handle_id to use the existing data_file_handle_id. The creator of the file must also be the owner of the data_file_handle_id to have permission to store the file.
external_url	(Optional[str]) The external URL of this file. If this is set AND synapse_store is FALSE, only a reference to this URL and the file metadata will be stored in Synapse. The file itself will not be uploaded. If this attribute is set it will override the path.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance. Activity cannot be removed during a store operation by setting it to NULL. You must use: synDelete or synDisassociateActivityFromEntity .

annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the file. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.
content_type	(Optional[str]) (New Upload Only) Used to manually specify Content-type header, for example 'application/png' or 'application/json; charset=UTF-8'. If not specified, the content type will be derived from the file extension. This can be specified only during the initial store of this file or any time there is a new file to upload. In order to change this after the File has been created use synChangeMetadata .
content_size	(Optional[int]) (New Upload Only) The size of the file in bytes. This can be specified only during the initial creation of the File. This is also only applicable to files not uploaded to Synapse. ie: <code>synapse_store</code> is <code>FALSE</code> .
content_md5	(Optional[str]) (Store only) The MD5 of the file is known. If not supplied this will be computed in the client is possible. If supplied for a file entity already stored in Synapse it will be calculated again to check if a new upload needs to occur. This will not be filled in during a read for data. It is only used during a store operation. To retrieve the md5 of the file after read from synapse use the <code>\$file_handle\$content_md5</code> attribute.
create_or_update	(bool) (Store only) Indicates whether the method should automatically perform an update if the file conflicts with an existing Synapse object.
force_version	(bool) (Store only) Indicates whether the method should increment the version of the object if something within the entity has changed. For example updating the description or name. You may set this to <code>FALSE</code> and an update to the entity will not increment the version. Updating the <code>version_label</code> attribute will also cause a version update regardless of this flag. An update to the MD5 of the file will force a version update regardless of this flag.
is_restricted	(bool) (Store only) If set to <code>TRUE</code> , an email will be sent to the Synapse access control team to start the process of adding terms-of-use or review board approval for this entity. You will be contacted with regards to the specific data being restricted and the requirements of access. This may be used only by an administrator of the specified file.
merge_existing_annotations	(bool) (Store only) Works in conjunction with <code>create_or_update</code> in that this is only evaluated if <code>create_or_update</code> is <code>TRUE</code> . If this entity exists in Synapse that has annotations that are not present in a store operation, these annotations will be added to the entity. If this is <code>FALSE</code> any annotations that are not present within a store operation will be removed from this entity. This allows one to complete a destructive update of annotations on an entity.
associate_activity_to_new_version	(bool) (Store only) Works in conjunction with <code>create_or_update</code> in that this is only evaluated if <code>create_or_update</code> is <code>TRUE</code> . When <code>TRUE</code> an activity already

attached to the current version of this entity will be associated the new version during a store operation if the version was updated. This is useful if you are updating the entity and want to ensure that the activity is persisted onto the new version the entity.

<code>synapse_store</code>	(bool) (Store only) Whether the File should be uploaded or if FALSE: only the path should be stored when <code>synStore</code> is called.
<code>download_file</code>	(bool) (Get only) If TRUE the file will be downloaded.
<code>if_collision</code>	(str) (Get only) Determines how to handle file collisions. Defaults to "keep.both". May be: - <code>overwrite.local</code> - <code>keep.local</code> - <code>keep.both</code>
<code>synapse_container_limit</code>	(Optional[str]) (Get only) A Synapse ID used to limit the search in Synapse if file is specified as a local file. That is, if the file is stored in multiple locations in Synapse only the ones in the specified folder/project will be returned.
<code>etag</code>	(Optional[str]) (Read Only) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
<code>created_on</code>	(Optional[str]) (Read Only) The date this entity was created.
<code>modified_on</code>	(Optional[str]) (Read Only) The date this entity was last modified.
<code>created_by</code>	(Optional[str]) (Read Only) The ID of the user that created this entity.
<code>modified_by</code>	(Optional[str]) (Read Only) The ID of the user that last modified this entity.
<code>version_number</code>	(Optional[int]) (Read Only) The version number issued to this version on the object.
<code>is_latest_version</code>	(Optional[bool]) (Read Only) If this is the latest version of the object.
<code>file_handle</code>	(Optional[FileHandle]) (Read Only) The file handle associated with this entity.

Format

An S3-style class object.

Value

NULL

Methods

- `File(id=NULL, name=NULL, path=NULL, description=NULL, parent_id=NULL, version_label=NULL, version_comment=NULL, data_file_handle_id=NULL, external_url=NULL, activity=NULL, annotations=list(), content_type=NULL, content_size=NULL, content_md5=NULL, create_or_update=TRUE, force_version=TRUE, is_restricted=FALSE, merge_existing_annotations=TRUE, associate_activity_to_synapse_store=TRUE, download_file=TRUE, if_collision="keep.both", synapse_container_limit=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL,`

`version_number=NULL, is_latest_version=NULL, file_handle=NULL)`: Constructor for `File`

- `synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)`: Bind a JSON schema to the entity.
- `synChangeMetadata(instance, name=NULL, download_as=NULL, content_type=NULL, synapse_client=NULL)`: Change File Entity metadata for properties that are immutable after creation through the store method.
- `synCopy(instance, parent_id, update_existing=FALSE, copy_annotations=TRUE, copy_activity="traceback", synapse_client=NULL)`: Copy the file to another Synapse location. Defaults to the latest version of the file, or the `version_number` specified in the instance.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetFromId(cls, synapse_id, synapse_client=NULL)`: Wrapper for `synGet`.
- `synGetFromPath(cls, path, synapse_client=NULL)`: Get the file from Synapse. If the path of the file matches multiple files within Synapse the first one found will be returned. The other matching files will be printed to the console.
Wrapper for `synGet`.
- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synGetSchema(instance, synapse_client=NULL)`: Get the JSON schema bound to the entity.
- `synGetSchemaDerivedKeys(instance, synapse_client=NULL)`: Retrieve derived JSON schema keys for the entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synSetPermissions`(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL): Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synUnbindSchema`(instance, synapse_client=NULL): Unbind the JSON schema bound to the entity.
- `synValidateSchema`(instance, synapse_client=NULL): Validate the entity against the bound JSON schema.

Examples

```
## Not run:
## Example 1: Uploading a file
library(synapser)
synLogin()

file = File(
  path = "/path/to/file.txt",
  parent_id = "syn123",
  description = "My data file",
  annotations = list(species = "Homo sapiens")
) |> synStore()
print(file$id)

## Example 2: Getting a file's metadata without downloading it
library(synapser)
synLogin()

file = File(id = "syn456") |> synGet(file_options = FileOptions(download_file = FALSE))
print(file$name)

## End(Not run)
```

FileOptions

FileOptions

Description

Configuration options specific to File entities when using the factory methods.

This dataclass allows you to customize how File entities are handled during retrieval, including download behavior, file location, and collision handling.

Usage

```
FileOptions(download_file=TRUE, download_location=NULL, if_collision="keep.both")
```

Arguments

- `download_file` (bool) Whether to automatically download the file content when retrieving the File entity. If TRUE, the file will be downloaded to the local filesystem. If FALSE, only the metadata will be retrieved. Default is TRUE.
- `download_location` (Optional[str]) The local directory path where the file should be downloaded. If NULL, the file will be downloaded to the default Synapse cache location. If specified, must be a valid directory path. Default is NULL.
- `if_collision` (str) Strategy to use when a file with the same name already exists at the download location. Valid options are:
- "keep.both": Keep both files by appending a number to the new file
 - "overwrite.local": Overwrite the existing local file
 - "keep.local": Keep the existing local file and skip download
- Default is "keep.both".

Format

An S3-style class object.

Value

NULL

Methods

- `FileOptions(download_file=TRUE, download_location=NULL, if_collision="keep.both")`: Constructor for [FileOptions](#)

Examples

```
## Not run:
# Configure file download options:
library(synapser)

# Download file to specific location with overwrite
file_options = FileOptions(
  download_file = TRUE,
  download_location = "/path/to/downloads/",
  if_collision = "overwrite.local"
)

# Only retrieve metadata, don't download file content
metadata_only = FileOptions(download_file = FALSE)

## End(Not run)
```

Folder	<i>Folder</i>
--------	---------------

Description

Folder is a hierarchical container for organizing data in Synapse.

Usage

```
Folder(id=NULL, name=NULL, parent_id=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=
```

Arguments

id	(Optional[str]) The unique immutable ID for this folder. A new ID will be generated for new Folders. Once issued, this ID is guaranteed to never change or be re-issued.
name	(Optional[str]) The name of this folder. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses.
parent_id	(Optional[str]) The ID of the Project or Folder that is the parent of this Folder.
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
etag	(Optional[str]) (Read Only) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
created_on	(Optional[str]) (Read Only) The date this entity was created.
modified_on	(Optional[str]) (Read Only) The date this entity was last modified.
created_by	(Optional[str]) (Read Only) The ID of the user that created this entity.
modified_by	(Optional[str]) (Read Only) The ID of the user that last modified this entity.
files	(list[File]) Files that exist within this folder.
folders	(list[Folder]) Folders that exist within this folder.
tables	(list[Table]) Tables that exist within this folder.
entityviews	(list[EntityView]) Entity views that exist within this folder.
submissionviews	(list[SubmissionView]) Submission views that exist within this folder.
datasets	(list[Dataset]) Datasets that exist within this folder.
datasetcollections	(list[DatasetCollection]) Dataset collections that exist within this folder.
materializedviews	(list[MaterializedView]) Materialized views that exist within this folder.
virtualtables	(list[VirtualTable]) Virtual tables that exist within this folder.
searchindexes	(list[SearchIndex]) Search indexes that exist within this folder.

annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the folder. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.
is_restricted	(bool) (Store only) If set to <code>TRUE</code> , an email will be sent to the Synapse access control team to start the process of adding terms-of-use or review board approval for this entity. You will be contacted with regards to the specific data being restricted and the requirements of access.
create_or_update	(bool) (Store only) Indicates whether the method should automatically perform an update if the resource conflicts with an existing Synapse object. When <code>TRUE</code> this means that any changes to the resource will be non-destructive. This boolean is ignored if you've already stored or retrieved the resource from Synapse for this instance at least once. Any changes to the resource will be destructive in this case. For example if you want to delete the content for a field you will need to call <code>.get()</code> and then modify the field.

Format

An S3-style class object.

Value

`NULL`

Methods

- `Folder(id=NULL, name=NULL, parent_id=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, files=list(), folders=list(), tables=list(), entityviews=list(), submissionviews=list(), datasets=list(), datasetcollections=list(), materializedviews=list(), virtualtables=list(), searchindexes=list(), annotations=list(), is_restricted=FALSE, create_or_update=TRUE)`: Constructor for [Folder](#)
- `synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)`: Bind a JSON schema to the entity.
- `synCopy(instance, parent_id, copy_annotations=TRUE, exclude_types=NULL, file_update_existing=FALSE, file_copy_activity="traceback", synapse_client=NULL)`: Copy the folder to another Synapse location. This will recursively copy all Tables, Links, Files, and Folders within the folder.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synDeleteProjectSetting(instance, setting_id, synapse_client=NULL)`: Delete a project setting by its setting ID.
- `synFlattenFileList(instance)`: Recursively loop over all of the already retrieved files and folders and return a list of all files in the container.
- `synGenerateSyncManifest(instance, directory_path, manifest_path, synapse_client=NULL)`: Walk a local directory, mirror its folder hierarchy under this container in Synapse, and write a CSV manifest ready for `synSyncToSynapse()`.
The manifest has two columns: `path` (absolute, symlink-resolved) and `parentId` (the Synapse ID of the file's containing folder). Existing Synapse folders with matching names and parents are reused. Directory symlinks inside `directory_path` are not followed; file symlinks record the symlink path and upload the target's contents. Zero-byte files are skipped with a warning — Synapse rejects empty files. I/O errors during walk are logged and skipped; an empty source directory produces a warning and a header-only manifest.
- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetInvalidValidation(instance, synapse_client=NULL)`: Get invalid JSON schema validation results for a container entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synGetProjectSetting(instance, setting_type="upload", synapse_client=NULL)`: Get the project setting for this entity.
- `synGetSchema(instance, synapse_client=NULL)`: Get the JSON schema bound to the entity.
- `synGetSchemaDerivedKeys(instance, synapse_client=NULL)`: Retrieve derived JSON schema keys for the entity.
- `synGetSchemaValidationStatistics(instance, synapse_client=NULL)`: Get validation statistics for a container entity.
- `synGetStsStorageToken(instance, permission, output_format="json", min_remaining_life=NULL, synapse_client=NULL)`: Get STS (AWS Security Token Service) credentials for direct access to the storage location backing this entity. These credentials can be used with AWS tools such as the AWS CLI.
- `synIndexFilesForMigration(instance, dest_storage_location_id, db_path=NULL, source_storage_location_id=NULL, file_version_strategy="new", include_table_files=FALSE, continue_on_error=FALSE, synapse_client=NULL)`: Index files in this entity for migration to a new storage location.

This is the first step in migrating files to a new storage location. After indexing, use `synMigrateIndexedFiles()` to perform the actual migration.

- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synMapDirectoryToAllContainedFiles(instance, root_path)`: Recursively loop over all of the already retrieved files and folders. Then return back a named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

This is used during the creation of the manifest TSV file during the `syncFromSynapse` utility function.

- `synMigrateIndexedFiles(instance, db_path, create_table_snapshots=TRUE, continue_on_error=FALSE, force=FALSE, synapse_client=NULL)`: Migrate files that have been indexed with `synIndexFilesForMigration()`.

This is the second step in migrating files to a new storage location. Files must first be indexed using `synIndexFilesForMigration()`.

Interactive confirmation: When called from an interactive shell and `force=FALSE` (the default), this function will print the number of items queued for migration and prompt for confirmation before proceeding. If standard output is not connected to an interactive terminal (e.g. a script or CI environment), migration is aborted unless `force=TRUE` is set.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSetStorageLocation(instance, storage_location_id=1L, synapse_client=NULL)`: Set the upload storage location for this entity. This configures where files uploaded to this entity will be stored.

This is a destructive update. The provided `storage_location_id` value(s) will **replace** any storage locations previously configured on this entity. To add a storage location without removing existing ones, first retrieve the current setting with `synGetProjectSetting()`, then pass its existing locations together with the new ID as `storage_location_id`. The first ID in the list is the default upload destination.

- `synSyncFromSynapse(instance, path=NULL, recursive=TRUE, download_file=TRUE, if_collision="overwrite", failure_strategy="LOG_EXCEPTION", include_activity=TRUE, follow_link=FALSE, link_hops=1L, queue=NULL, include_types=NULL, manifest="all", synapse_client=NULL)`: Sync this container and all possible sub-folders from Synapse. By default this will download the files that are found and it will populate the files and folders attributes with the found files and folders, along with all other entity types (tables, entityviews, etc.) present in the container. If you only want to retrieve the full tree of metadata about your container specify `download_file` as `FALSE`.

This works similar to the Python `synapseutils.syncFromSynapse` utility function, and generates a `manifest.csv` file in each synced directory. The manifest uses CSV format with `parentId` and `ID` columns, interoperable with the Synapse UI download cart and `synDownloadListFiles()` output.

Supports syncing Files, Folders, Tables, EntityViews, SubmissionViews, Datasets, Dataset-Collections, MaterializedViews, and VirtualTables from Synapse. The metadata for these entity types will be populated in their respective attributes (`files`, `folders`, `tables`, `entityviews`, `submissionviews`, `datasets`, `datasetcollections`, `materializedviews`, `virtualtables`) if they are found within the container.

- `synSyncToSynapse(instance, manifest_path, dry_run=FALSE, send_messages=TRUE, retries=4L, merge_existing_annotations=TRUE, associate_activity_to_new_version=FALSE, synapse_client=NULL)`: Upload files to Synapse using a manifest CSV file.
Accepts manifests produced by `synSyncFromSynapse()`, `synDownloadListFiles()`, or the Synapse UI download cart. The manifest must have at minimum a `path` and `parentId` column. All other columns that are not part of the standard manifest column set are treated as file annotations.
Standard manifest columns: `ID`, `name`, `parentId`, `contentType`, `path`, `synapseStore`, `activityName`, `activityDescription`, `forceVersion`, `used`, `executed`
- `synUnbindSchema(instance, synapse_client=NULL)`: Unbind the JSON schema bound to the entity.
- `synValidateSchema(instance, synapse_client=NULL)`: Validate the entity against the bound JSON schema.
- `synWalk(instance, follow_link=FALSE, include_types=NULL, recursive=TRUE, display_ascii_tree=FALSE, synapse_client=NULL)`: Traverse through the hierarchy of entities stored under this container. Mimics `os.walk()` behavior but yields `EntityHeader` objects, with optional ASCII tree display that continues printing as the walk progresses.

Examples

```
## Not run:
## Example 1: Creating a folder
library(synapser)
synLogin()

folder = Folder(name = "My Folder", parent_id = "syn123") |> synStore()
print(folder$id)

## Example 2: Listing the contents of a folder
library(synapser)
synLogin()

folder = Folder(id = "syn456") |> synSyncFromSynapse(download_file = FALSE, recursive = FALSE)
for (file in folder$files) {
  print(file$name)
}

## End(Not run)
```

Link	<i>Link</i>
------	-------------

Description

A Link entity within Synapse that references another entity.
Represents a [Synapse Link](#).

Usage

Link(name=NULL, description=NULL, id=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=N

Arguments

name	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
id	(Optional[str]) The unique immutable ID for this entity. A new ID will be generated for new Entities. Once issued, this ID is guaranteed to never change or be re-issued
etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client’s current representation of an entity is out-of-date.
created_on	(Optional[str]) (Read Only) The date this entity was created.
modified_on	(Optional[str]) (Read Only) The date this entity was last modified.
created_by	(Optional[str]) (Read Only) The ID of the user that created this entity.
modified_by	(Optional[str]) (Read Only) The ID of the user that last modified this entity.
parent_id	(Optional[str]) The ID of the Entity that is the parent of this Entity.
target_id	(Optional[str]) The ID of the entity to which this link refers
target_version_number	(Optional[int]) The version number of the entity to which this link refers
links_to_class_name	(Optional[str]) The synapse Entity’s class name that this link points to.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance. Activity cannot be removed during a store operation by setting it to NULL. You must use synDelete or synDisassociateActivityFromEntity .
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the link. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list list() or NULL and store the entity.

`concrete_type` (Optional[str]) Indicates which implementation of Entity this object represents.
The value is the fully qualified class name, e.g. `org.sagebionetworks.repo.model.FileEntity`.

Format

An S3-style class object.

Value

NULL

Methods

- `Link(name=NULL, description=NULL, id=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, parent_id=NULL, target_id=NULL, target_version_number=NULL, links_to_class_name=NULL, activity=NULL, annotations=list(), concrete_type=NULL):`
Constructor for [Link](#)

Examples

```
## Not run:
## Creating a link to another entity
library(synapser)
synLogin()

link = Link(name = "My Link", parent_id = "syn123", target_id = "syn456") |> synStore()
print(link$id)

# Link to a specific version of the target
link = Link(
  name = "My Versioned Link",
  parent_id = "syn123",
  target_id = "syn456",
  target_version_number = 2L
) |> synStore()

## End(Not run)
```

LinkOptions

LinkOptions

Description

Configuration options specific to Link entities when using the factory methods.

This dataclass controls how Link entities are handled during retrieval, particularly whether the link should be followed to return the target entity or if the Link entity itself should be returned.

Usage

```
LinkOptions(follow_link=TRUE)
```

Arguments

`follow_link` (bool) Whether to follow the link and return the target entity instead of the Link entity itself. If TRUE, the factory method will return the entity that the Link points to (e.g., if a Link points to a File, a File object will be returned). If FALSE, the Link entity itself will be returned, allowing you to inspect the link's properties such as `target_id`, `target_version`, etc. Default is TRUE.

Format

An S3-style class object.

Value

NULL

Methods

- `LinkOptions(follow_link=TRUE)`: Constructor for [LinkOptions](#)

Note

- When `follow_link=TRUE`, the returned entity type depends on what the Link points to (could be File, Project, Folder, etc.)
- When `follow_link=FALSE`, a Link entity is always returned

Examples

```
## Not run:
# Configure link following behavior:
library(synapser)

# Follow the link and return the target entity
follow_target = LinkOptions(follow_link = TRUE)

# Return the Link entity itself
return_link = LinkOptions(follow_link = FALSE)

## End(Not run)
```

MaterializedView

MaterializedView

Description

A materialized view is a type of table that is automatically built from a Synapse SQL query. Its content is read only and based off the `defining_sql` attribute. The SQL of the materialized view may contain JOIN clauses on multiple tables.

A `MaterializedView` object represents this concept in Synapse: <<https://rest-docs.synapse.org/rest/org/sagebionetworks/rep>

Usage

MaterializedView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=

Arguments

id	(Optional[str]) The unique immutable ID for this entity. Once issued, this ID is guaranteed to never change or be re-issued.
name	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses.
parent_id	(Optional[str]) The ID of the Entity that is the parent of this entity.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse.
version_number	(Optional[int]) The version number issued to this version on the object.
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
created_on	(Optional[str]) The date this entity was created.
modified_on	(Optional[str]) The date this entity was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format.
created_by	(Optional[str]) The ID of the user that created this entity.
modified_by	(Optional[str]) The ID of the user that last modified this entity.
version_label	(Optional[str]) The version label for this entity.
version_comment	(Optional[str]) The version comment for this entity.
is_latest_version	(Optional[bool]) If this is the latest version of the object.
columns	(Optional[OrderedDict]) (Read Only) The columns of a materialized view are dynamic based on the select statement of the definingSQL. This list of columnIds is for read-only purposes.
is_search_enabled	(Optional[bool]) When creating or updating a table or view specifies if full text search should be enabled.
defining_sql	(Optional[str]) The synapse SQL statement that defines the data in the materialized view.
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the materialized view. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list list() or NULL and store the entity.

Format

An S3-style class object.

Value

NULL

Methods

- `MaterializedView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, columns=list(), is_search_enabled=NULL, defining_sql=NULL, annotations=list())`: Constructor for [MaterializedView](#)
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a `DataFrame` or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API:

<<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

Examples

```
## Not run:
## Example 1: Create a new materialized view with a defining SQL query.
library(synapser)
synLogin()

materialized_view = MaterializedView(
  name = "My Materialized View",
  description = "A test materialized view",
  parent_id = "syn12345",
  defining_sql = "SELECT * FROM syn67890"
) |> synStore()
print(sprintf("Created Materialized View with ID: %s", materialized_view$id))

## Example 2: Update the defining SQL of an existing materialized view.
library(synapser)
synLogin()

materialized_view = MaterializedView(id = "syn12345") |>
  synGet() |>
  synSetProperties(defining_sql = "SELECT column1, column2 FROM syn67890") |>
  synStore()
print("Updated Materialized View defining SQL.")

## Example 3: Delete a materialized view.
library(synapser)
```

```

synLogin()

MaterializedView(id = "syn12345") |> synDelete()
print("Deleted Materialized View.")

## Example 4: Query data from a materialized view.
library(synapser)
synLogin()

query_result = synQuery(query = "SELECT * FROM syn12345")
print(query_result)

## Example 5: Retrieve and update annotations for a materialized view.
library(synapser)
synLogin()

materialized_view = MaterializedView(id = "syn12345") |> synGet()
materialized_view = materialized_view |>
  synSetProperties(
    annotations = append(
      materialized_view$annotations,
      list(key1 = "value1", key2 = "value2")
    )
  ) |>
  synStore()
print("Updated annotations for Materialized View.")

## Example 6: Create a materialized view with a JOIN clause.
library(synapser)
synLogin()

defining_sql = "
SELECT t1.column1 AS new_column1, t2.column2 AS new_column2
FROM syn12345 t1
JOIN syn67890 t2
ON t1.id = t2.foreign_id
"

materialized_view = MaterializedView(
  name = "Join Materialized View",
  description = "A materialized view with a JOIN clause",
  parent_id = "syn11111",
  defining_sql = defining_sql
) |> synStore()
print(sprintf("Created Materialized View with ID: %s", materialized_view$id))

## Example 7: Create a materialized view with a LEFT JOIN clause.
library(synapser)
synLogin()

defining_sql = "
SELECT t1.column1 AS new_column1, t2.column2 AS new_column2
FROM syn12345 t1

```

```

LEFT JOIN syn67890 t2
ON t1.id = t2.foreign_id
"

materialized_view = MaterializedView(
    name = "Left Join Materialized View",
    description = "A materialized view with a LEFT JOIN clause",
    parent_id = "syn11111",
    defining_sql = defining_sql
) |> synStore()
print(sprintf("Created Materialized View with ID: %s", materialized_view$id))

## Example 8: Create a materialized view with a RIGHT JOIN clause.
library(synapser)
synLogin()

defining_sql = "
SELECT t1.column1 AS new_column1, t2.column2 AS new_column2
FROM syn12345 t1
RIGHT JOIN syn67890 t2
ON t1.id = t2.foreign_id
"

materialized_view = MaterializedView(
    name = "Right Join Materialized View",
    description = "A materialized view with a RIGHT JOIN clause",
    parent_id = "syn11111",
    defining_sql = defining_sql
) |> synStore()
print(sprintf("Created Materialized View with ID: %s", materialized_view$id))

## Example 9: Create a materialized view with a UNION clause.
library(synapser)
synLogin()

defining_sql = "
SELECT column1 AS new_column1, column2 AS new_column2
FROM syn12345
UNION
SELECT column1 AS new_column1, column2 AS new_column2
FROM syn67890
"

materialized_view = MaterializedView(
    name = "Union Materialized View",
    description = "A materialized view with a UNION clause",
    parent_id = "syn11111",
    defining_sql = defining_sql
) |> synStore()
print(sprintf("Created Materialized View with ID: %s", materialized_view$id))

## End(Not run)

```

Organization

Organization

Description

Represents an **Organization**.

Usage

```
Organization(name=NULL, id=NULL, created_on=NULL, created_by=NULL)
```

Arguments

name	(Optional[character]) The name of the organization
id	(Optional[character]) The ID of the organization
created_on	(Optional[character]) The date this organization was created
created_by	(Optional[character]) The ID of the user that created this organization

Format

An S3-style class object.

Value

NULL

Methods

- `Organization(name=NULL, id=NULL, created_on=NULL, created_by=NULL)`: Constructor for [Organization](#)
- `synGetAcl(instance, synapse_client=NULL)`: Gets the ACL for this organization
- `synGetJsonSchemas(instance, synapse_client=NULL)`: Gets the JSON Schemas that are part of this organization
- `synUpdateAcl(instance, principal_id, access_type, synapse_client=NULL)`: Updates the ACL for a principal for this organization

Examples

```
## Not run:
## Creating a JSON schema organization
library(synapser)
synLogin()

org = Organization(name = "myUniqueOrgName") |> synStore()
print(org$id)

# List the JSON schemas in the organization
```

```
schemas = org |> synGetJsonSchemas()

## End(Not run)
```

Project

Project

Description

A Project is a top-level container for organizing data in Synapse.

Usage

```
Project(id=NULL, name=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_b
```

Arguments

id	(Optional[str]) The unique immutable ID for this project. A new ID will be generated for new Projects. Once issued, this ID is guaranteed to never change or be re-issued
name	(Optional[str]) The name of this project. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
created_on	(Optional[str]) The date this entity was created.
modified_on	(Optional[str]) The date this entity was last modified.
created_by	(Optional[str]) The ID of the user that created this entity.
modified_by	(Optional[str]) The ID of the user that last modified this entity.
alias	(Optional[str]) The project alias for use in friendly project urls.
files	(list[File]) Any files that are at the root directory of the project.
folders	(list[Folder]) Any folders that are at the root directory of the project.
tables	(list[Table]) Any tables that are at the root directory of the project.
entityviews	(list[EntityView]) Any entity views that are at the root directory of the project.
submissionviews	(list[SubmissionView]) Any submission views that are at the root directory of the project.
datasets	(list[Dataset]) Any datasets that are at the root directory of the project.
datasetcollections	(list[DatasetCollection]) Any dataset collections that are at the root directory of the project.

materializedviews	(list[MaterializedView]) Any materialized views that are at the root directory of the project.
virtualtables	(list[VirtualTable]) Any virtual tables that are at the root directory of the project.
searchindexes	(list[SearchIndex]) Any search indexes that are at the root directory of the project.
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the project. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.
create_or_update	(bool) (Store only) Indicates whether the method should automatically perform an update if the resource conflicts with an existing Synapse object. When <code>TRUE</code> this means that any changes to the resource will be non-destructive. This boolean is ignored if you've already stored or retrieved the resource from Synapse for this instance at least once. Any changes to the resource will be destructive in this case. For example if you want to delete the content for a field you will need to call <code>synGet()</code> and then modify the field.
parent_id	(Optional[str]) The parent ID of the project. In practice projects do not have a parent, but this is required for the inner workings of Synapse.

Format

An S3-style class object.

Value

`NULL`

Methods

- `Project(id=NULL, name=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, alias=NULL, files=list(), folders=list(), tables=list(), entityviews=list(), submissionviews=list(), datasets=list(), datasetcollections=list(), materializedviews=list(), virtualtables=list(), searchindexes=list(), annotations=list(), create_or_update=TRUE, parent_id=NULL)`: Constructor for [Project](#)
- `synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)`: Bind a JSON schema to the entity.
- `synCopy(instance, destination_id, copy_annotations=TRUE, copy_wiki=TRUE, exclude_types=NULL, file_update_existing=FALSE, file_copy_activity="traceback", synapse_client=NULL)`: You must have already created the Project you will be copying to. It will have its own Synapse ID and unique name that you will use as the `destination_id`.

Copy the project to another Synapse project. This will recursively copy all Tables, Links, Files, and Folders within the project.

- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synDeleteProjectSetting(instance, setting_id, synapse_client=NULL)`: Delete a project setting by its setting ID.
- `synFlattenFileList(instance)`: Recursively loop over all of the already retrieved files and folders and return a list of all files in the container.
- `synGenerateSyncManifest(instance, directory_path, manifest_path, synapse_client=NULL)`: Walk a local directory, mirror its folder hierarchy under this container in Synapse, and write a CSV manifest ready for `synSyncToSynapse()`.

The manifest has two columns: `path` (absolute, symlink-resolved) and `parentId` (the Synapse ID of the file's containing folder). Existing Synapse folders with matching names and parents are reused. Directory symlinks inside `directory_path` are not followed; file symlinks record the symlink path and upload the target's contents. Zero-byte files are skipped with a warning — Synapse rejects empty files. I/O errors during walk are logged and skipped; an empty source directory produces a warning and a header-only manifest.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetInvalidValidation(instance, synapse_client=NULL)`: Get invalid JSON schema validation results for the child entities of a container entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synGetProjectSetting(instance, setting_type="upload", synapse_client=NULL)`: Get the project setting for this entity.
- `synGetSchema(instance, synapse_client=NULL)`: Get the JSON schema bound to the entity.
- `synGetSchemaDerivedKeys(instance, synapse_client=NULL)`: Retrieve derived JSON schema keys for the entity.

- `synGetSchemaValidationStatistics(instance, synapse_client=NULL)`: Get validation statistics for a container entity.
- `synGetStsStorageToken(instance, permission, output_format="json", min_remaining_life=NULL, synapse_client=NULL)`: Get STS (AWS Security Token Service) credentials for direct access to the storage location backing this entity. These credentials can be used with AWS tools such as the AWS CLI.
- `synIndexFilesForMigration(instance, dest_storage_location_id, db_path=NULL, source_storage_location_id=NULL, file_version_strategy="new", include_table_files=FALSE, continue_on_error=FALSE, synapse_client=NULL)`: Index files in this entity for migration to a new storage location.

This is the first step in migrating files to a new storage location. After indexing, use `synMigrateIndexedFiles()` to perform the actual migration.

- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synMapDirectoryToAllContainedFiles(instance, root_path)`: Recursively loop over all of the already retrieved files and folders. Then return back a named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

This is used during the creation of the manifest TSV file during the `syncFromSynapse` utility function.

- `synMigrateIndexedFiles(instance, db_path, create_table_snapshots=TRUE, continue_on_error=FALSE, force=FALSE, synapse_client=NULL)`: Migrate files that have been indexed with `synIndexFilesForMigration()`.

This is the second step in migrating files to a new storage location. Files must first be indexed using `synIndexFilesForMigration()`.

Interactive confirmation: When called from an interactive shell and `force=FALSE` (the default), this function will print the number of items queued for migration and prompt for confirmation before proceeding. If standard output is not connected to an interactive terminal (e.g. a script or CI environment), migration is aborted unless `force=TRUE` is set.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSetStorageLocation(instance, storage_location_id=1L, synapse_client=NULL)`: Set the upload storage location for this entity. This configures where files uploaded to this entity will be stored.

This is a destructive update. The provided `storage_location_id` value(s) will **replace** any storage locations previously configured on this entity. To add a storage location without removing existing ones, first retrieve the current setting with `synGetProjectSetting()`, then pass its existing locations together with the new ID as `storage_location_id`. The first ID in the list is the default upload destination.

- `synSyncFromSynapse(instance, path=NULL, recursive=TRUE, download_file=TRUE, if_collision="overwrite", failure_strategy="LOG_EXCEPTION", include_activity=TRUE, follow_link=FALSE, link_hops=1L, queue=NULL, include_types=NULL, manifest="all", synapse_client=NULL)`: Sync this container and all possible sub-folders from Synapse. By default this will download the files that are found and it will populate the files and folders attributes with the found files and folders, along with all other entity types (tables, entityviews, etc.) present in the container. If you only want to retrieve the full tree of metadata about your container specify `download_file` as `FALSE`.

This works similarly to the legacy `synapseutils.syncFromSynapse` utility function, and generates a `manifest.csv` file in each synced directory. The manifest uses CSV format with `parentId` and `ID` columns, interoperable with the Synapse UI download cart and `synDownloadListFiles()` output.

Supports syncing Files, Folders, Tables, EntityViews, SubmissionViews, Datasets, Dataset-Collections, MaterializedViews, and VirtualTables from Synapse. The metadata for these entity types will be populated in their respective attributes (`files`, `folders`, `tables`, `entityviews`, `submissionviews`, `datasets`, `datasetcollections`, `materializedviews`, `virtualtables`) if they are found within the container.

- `synSyncToSynapse(instance, manifest_path, dry_run=FALSE, send_messages=TRUE, retries=4L, merge_existing_annotations=TRUE, associate_activity_to_new_version=FALSE, synapse_client=NULL)`: Upload files to Synapse using a manifest CSV file.

Accepts manifests produced by `synSyncFromSynapse()`, `synDownloadListFiles()`, or the Synapse UI download cart. The manifest must have at minimum a `path` and `parentId` column. All other columns that are not part of the standard manifest column set are treated as file annotations.

Standard manifest columns: `ID`, `name`, `parentId`, `contentType`, `path`, `synapseStore`, `activityName`, `activityDescription`, `forceVersion`, `used`, `executed`

- `synUnbindSchema(instance, synapse_client=NULL)`: Unbind the JSON schema bound to the entity.
- `synValidateSchema(instance, synapse_client=NULL)`: Validate the entity against the bound JSON schema.
- `synWalk(instance, follow_link=FALSE, include_types=NULL, recursive=TRUE, display_ascii_tree=FALSE, synapse_client=NULL)`: Traverse through the hierarchy of entities stored under this container. Mimics `os.walk()` behavior but yields `EntityHeader` objects, with optional ASCII tree display that continues printing as the walk progresses.

Examples

```
## Not run:
```

```
## Example 1: Creating a project
# This example shows how to create a project
library(synapser)
synLogin()

my_annotations = list(
  my_single_key_string = "a",
  my_key_string = list("b", "a", "c")
)
project = Project(
  name = "My unique project name",
  annotations = my_annotations,
  description = "This is a project with random data."
)

project = project |> synStore()

project

## Example 2: Storing several files to a project
# This example shows how to store several files to a project
file_1 = File(
  path = path_to_file_1,
  name = name_of_file_1
)
file_2 = File(
  path = path_to_file_2,
  name = name_of_file_2
)
project$files = list(file_1, file_2)
project = project |> synStore()

## End(Not run)
```

StorageLocation

StorageLocation

Description

A storage location setting describes where files are uploaded to and downloaded from via Synapse. Storage location settings may be created for external locations, such as user-owned Amazon S3 buckets, Google Cloud Storage buckets, SFTP servers, or proxy storage.

Usage

```
StorageLocation(storage_location_id=NULL, storage_type=NULL, concrete_type=NULL, banner=NULL, descrip
```

Arguments

storage_location_id	(Optional[integer]) (Read Only) The unique ID for this storage location, assigned by the server on creation.
storage_type	(Optional[StorageLocationType]) The type of storage location, a StorageLocationType . Required when creating a new storage location via synStore() . Determines the concreteType sent to the Synapse REST API.
concrete_type	(Optional[character]) The concrete type of the storage location indicating which implementation this object represents.
banner	(Optional[character]) The banner text to display to a user every time a file is uploaded. This field is optional.
description	(Optional[character]) A description of the storage location. This description is shown when a user has to choose which upload destination to use.
bucket	(Optional[character]) The name of the S3 or Google Cloud Storage bucket. Applicable to SYNAPSE_S3, EXTERNAL_S3, EXTERNAL_GOOGLE_CLOUD, and EXTERNAL_OBJECT_STORE types.
base_key	(Optional[character]) The optional base key (prefix/folder) within the bucket. Applicable to SYNAPSE_S3, EXTERNAL_S3, and EXTERNAL_GOOGLE_CLOUD types.
sts_enabled	(Optional[logical]) Whether STS (AWS Security Token Service) is enabled on this storage location. Applicable to SYNAPSE_S3 and EXTERNAL_S3 types.
endpoint_url	(Optional[character]) The endpoint URL of the S3 service. Applicable to EXTERNAL_S3 (default: https://s3.amazonaws.com) and EXTERNAL_OBJECT_STORE types.
url	(Optional[character]) The base URL for uploading to the external destination. Applicable to EXTERNAL_SFTP type.
supports_subfolders	(Optional[logical]) Whether the destination supports creating subfolders under the base url. Applicable to EXTERNAL_SFTP type. Default: FALSE.
proxy_url	(Optional[character]) The HTTPS URL of the proxy used for upload and download. Applicable to PROXY type.
secret_key	(Optional[character]) The encryption key used to sign all pre-signed URLs used to communicate with the proxy. Applicable to PROXY type.
benefactor_id	(Optional[character]) An Entity ID (such as a Project ID). When set, any user with the 'create' permission on the given benefactorId will be allowed to create ProxyFileHandle using its storage location ID. Applicable to PROXY type.
upload_type	(Optional[UploadType]) (Read Only) The upload type for this storage location. Automatically derived from storage_type.
etag	(Optional[character]) (Read Only) Synapse employs an Optimistic Concurrency Control (OCC) scheme. The E-Tag changes every time the setting is updated.
created_on	(Optional[character]) (Read Only) The date this storage location setting was created.
created_by	(Optional[integer]) (Read Only) The ID of the user that created this storage location setting.

Format

An S3-style class object.

Value

NULL

Methods

- `StorageLocation(storage_location_id=NULL, storage_type=NULL, concrete_type=NULL, banner=NULL, description=NULL, bucket=NULL, base_key=NULL, sts_enabled=FALSE, endpoint_url="https://s3.amazonaws.com", url=NULL, supports_subfolders=FALSE, proxy_url=NULL, secret_key=NULL, benefactor_id=NULL, upload_type=NULL, etag=NULL, created_on=NULL, created_by=NULL)`: Constructor for [StorageLocation](#)

Examples

```
## Not run:
## Example 1: Creating an external S3 storage location
# Create a storage location backed by your own S3 bucket:
library(synapser)
synLogin()

storage = StorageLocation(
  storage_type = StorageLocationType(
    name = "EXTERNAL_S3",
    concrete_type = "ExternalS3StorageLocationSetting"
  ),
  bucket = "my-external-synapse-bucket",
  base_key = "path/within/bucket"
) |> synStore()

print(sprintf("Storage location ID: %s", storage$storage_location_id))

## Example 2: Creating a Google Cloud storage location
# Create a storage location backed by your own GCS bucket:
library(synapser)
synLogin()

storage = StorageLocation(
  storage_type = StorageLocationType(
    name = "EXTERNAL_GOOGLE_CLOUD",
    concrete_type = "ExternalGoogleCloudStorageLocationSetting"
  ),
  bucket = "my-gcs-bucket",
  base_key = "path/within/bucket"
) |> synStore()

## Example 3: Retrieving a storage location by ID
# Only the creator of a storage location can retrieve it by its ID.
library(synapser)
```

```
synLogin()

storage = StorageLocation(storage_location_id = 12345L) |> synGet()
print(sprintf("Type: %s, Bucket: %s", storage$storage_type$name, storage$bucket))

## End(Not run)
```

StorageLocationType	<i>StorageLocationType</i>
---------------------	----------------------------

Description

Describes a Synapse storage location type.

Each instance is a distinct object identified by its name, so SFTP and HTTPS remain separate even though they share the same backend concreteType (ExternalStorageLocationSetting).

Usage

```
StorageLocationType(name, concrete_type)
```

Arguments

- name (character) Human-readable identifier (e.g. "EXTERNAL_SFTP").
- concrete_type (character) The concreteType suffix sent to the Synapse REST API.

Format

An S3-style class object.

Value

NULL

Supported storage location types

A StorageLocationType is identified by both its name and its concrete_type, so construct it with one of the following pairs:

name	concrete_type
"SYNAPSE_S3"	"S3StorageLocationSetting"
"EXTERNAL_S3"	"ExternalS3StorageLocationSetting"
"EXTERNAL_GOOGLE_CLOUD"	"ExternalGoogleCloudStorageLocationSetting"
"EXTERNAL_SFTP"	"ExternalStorageLocationSetting"
"EXTERNAL_HTTPS"	"ExternalStorageLocationSetting"
"EXTERNAL_OBJECT_STORE"	"ExternalObjectStorageLocationSetting"
"PROXY"	"ProxyStorageLocationSettings"

Methods

- `StorageLocationType(name, concrete_type)`: Constructor for [StorageLocationType](#)

Examples

```
## Not run:
## Use a storage location type when creating a storage location
library(synapser)
synLogin()

external_s3 = StorageLocationType(
  name = "EXTERNAL_S3",
  concrete_type = "ExternalS3StorageLocationSetting"
)
storage = StorageLocation(
  storage_type = external_s3,
  bucket = "my-external-synapse-bucket"
) |> synStore()

## End(Not run)
```

StoreContainerOptions *StoreContainerOptions*

Description

Options for storing container entities (Project, Folder).

Usage

```
StoreContainerOptions(failure_strategy=NULL)
```

Arguments

`failure_strategy`
 (Optional[str]) Strategy for handling failures when storing child entities. One of "LOG_EXCEPTION" or "RAISE_EXCEPTION".

Format

An S3-style class object.

Value

NULL

Methods

- `StoreContainerOptions(failure_strategy=NULL)`: Constructor for [StoreContainerOptions](#)

Examples

```
## Not run:
## Using this class
# Stop storing a folder's children as soon as one of them fails
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
folder = folder |> synStore(
  container_options = StoreContainerOptions(failure_strategy = "RAISE_EXCEPTION")
)

## End(Not run)
```

StoreFileOptions

*StoreFileOptions***Description**

Options for storing File entities.

Usage

```
StoreFileOptions(synapse_store=NULL, content_type=NULL, merge_existing_annotations=NULL, associate_a
```

Arguments

`synapse_store` (Optional[bool]) Whether to store the file in Synapse or use an external URL.

`content_type` (Optional[str]) The content type of the file.

`merge_existing_annotations` (Optional[bool]) If TRUE, merge existing annotations with new ones. If FALSE, replace all annotations.

`associate_activity_to_new_version` (Optional[bool]) If TRUE, associate activity with new version. If FALSE, do not associate activity.

Format

An S3-style class object.

Value

NULL

Methods

- `StoreFileOptions(synapse_store=NULL, content_type=NULL, merge_existing_annotations=NULL, associate_activity_to_new_version=NULL)`: Constructor for [StoreFileOptions](#)

Examples

```
## Not run:
## Using this class
library(synapser)
synLogin()

file = File(path = "/path/to/file.txt", parent_id = "syn123")
file = file |> synStore(
  file_options = StoreFileOptions(
    content_type = "text/plain",
    merge_existing_annotations = TRUE
  )
)

## End(Not run)
```

StoreGridOptions	<i>StoreGridOptions</i>
------------------	-------------------------

Description

Options for storing Grid entities.

Usage

```
StoreGridOptions(attach_to_previous_session=FALSE, timeout=120L)
```

Arguments

attach_to_previous_session	(bool) If TRUE and using record_set_id, will attach to an existing active session if one exists.
timeout	(int) The number of seconds to wait for the job to complete.

Format

An S3-style class object.

Value

NULL

Methods

- `StoreGridOptions(attach_to_previous_session=FALSE, timeout=120L)`: Constructor for [StoreGridOptions](#)

Examples

```
## Not run:
## Using this class
# Pass to synStore() through the grid_options argument when storing a Grid
grid_options = StoreGridOptions(attach_to_previous_session = TRUE, timeout = 300L)

## End(Not run)
```

StoreJSONSchemaOptions

StoreJSONSchemaOptions

Description

Options for storing JSONSchema entities.

Usage

```
StoreJSONSchemaOptions(schema_body, version=NULL, dry_run=FALSE)
```

Arguments

schema_body	(dict) The body of the JSONSchema to store, provided as a named list.
version	(Optional[str]) The version of the JSONSchema body to store.
dry_run	(bool) Whether or not to do a dry-run.

Format

An S3-style class object.

Value

NULL

Methods

- `StoreJSONSchemaOptions(schema_body, version=NULL, dry_run=FALSE)`: Constructor for [StoreJSONSchemaOptions](#)

Examples

```
## Not run:
## Using this class
library(synapser)
synLogin()

schema_definition = list(
  "$id" = "mySchema",
```

```

        type = "object",
        properties = list(foo = list(type = "string")),
        required = list("foo")
    )
    schema = JSONSchema(name = "my.schema", organization_name = "myUniqueOrgName") |> synStore(
        json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = "0.0.1")
    )

## End(Not run)

```

StoreTableOptions	<i>StoreTableOptions</i>
-------------------	--------------------------

Description

Options for storing Table-like entities.

Usage

```
StoreTableOptions(dry_run=FALSE, job_timeout=600L)
```

Arguments

dry_run	(bool) If TRUE, will not actually store but will log what would be done.
job_timeout	(int) Maximum time to wait for table schema update job to complete.

Format

An S3-style class object.

Value

NULL

Methods

- `StoreTableOptions(dry_run=FALSE, job_timeout=600L)`: Constructor for [StoreTableOptions](#)

Examples

```

## Not run:
## Using this class
# Preview a table schema change without applying it
library(synapser)
synLogin()

table = Table(id = "syn123") |> synGet()
table |> synAddColumn(Column(name = "new_column", column_type = "STRING"))
table |> synStore(table_options = StoreTableOptions(dry_run = TRUE))

## End(Not run)

```

Submission	<i>Submission</i>
------------	-------------------

Description

A Submission object represents a Synapse Submission, which is created when a user submits an entity to an evaluation queue. <https://rest-docs.synapse.org/rest/org/sagebionetworks/evaluation/model/Submission.html>

Usage

Submission(id=NULL, user_id=NULL, submitter_alias=NULL, entity_id=NULL, version_number=NULL, evaluation_id=NULL)

Arguments

- id (Optional[character]) The unique ID of this Submission.
- user_id (Optional[character]) The ID of the user that submitted this Submission.
- submitter_alias (Optional[character]) The name of the user that submitted this Submission.
- entity_id (Optional[character]) The ID of the entity being submitted.
- version_number (Optional[integer]) The version number of the entity at submission.
- evaluation_id (Optional[character]) The ID of the Evaluation to which this Submission belongs.
- evaluation_round_id (Optional[character]) The ID of the EvaluationRound to which this was submitted (auto-filled upon creation).
- name (Optional[character]) The name of this Submission.
- created_on (Optional[character]) The date this Submission was created.
- team_id (Optional[character]) The ID of the team that submitted this submission (if it's a team submission).
- contributors (list[character]) User IDs of team members who contributed to this submission (if it's a team submission).
- submission_status (Optional[named list]) The status of this Submission.
- entity_bundle_json (Optional[character]) The bundled entity information at submission. This includes the entity, annotations, file handles, and other metadata.
- docker_repository_name (Optional[character]) For Docker repositories, the repository name.
- docker_digest (Optional[character]) For Docker repositories, the digest of the submitted Docker image.
- docker_tag (Optional[character]) For Docker repositories, the tag of the submitted Docker image.
- etag (Optional[character]) The current eTag of the Entity being submitted. If not provided, it will be automatically retrieved.

Format

An S3-style class object.

Value

NULL

Methods

- `Submission(id=NULL, user_id=NULL, submitter_alias=NULL, entity_id=NULL, version_number=NULL, evaluation_id=NULL, evaluation_round_id=NULL, name=NULL, created_on=NULL, team_id=NULL, contributors=list(), submission_status=NULL, entity_bundle_json=NULL, docker_repository_name=NULL, docker_digest=NULL, docker_tag=NULL, etag=NULL)`: Constructor for [Submission](#)
- `synCancel(instance, synapse_client=NULL)`: Cancel a Submission. Only the user who created the Submission may cancel it.
- `synGetEvaluationSubmissions(cls, evaluation_id, status=NULL, synapse_client=NULL)`: Retrieves all Submissions for a specified Evaluation queue.
- `synGetSubmissionCount(evaluation_id, synapse_client=NULL)`: Gets the number of Submissions for a specified Evaluation queue.
- `synGetUserSubmissions(cls, evaluation_id, synapse_client=NULL)`: Retrieves all user Submissions for a specified Evaluation queue.

Examples

```
## Not run:
## Example 1: Create and store a new Submission.
library(synapser)
synLogin()

# Create a new submission
submission = Submission(
  entity_id = "syn123456",
  evaluation_id = "9999999",
  name = "My Submission"
)

# Store the submission
stored_submission = submission |> synStore()
print(sprintf("Created submission with ID: %s", stored_submission$id))

## Example 2: Retrieve a Submission.
library(synapser)
synLogin()

submission = Submission(id = "9999999") |> synGet()
print(submission)

## Example 3: Get all submissions for a user.
library(synapser)
synLogin()
```

```

# Get submissions for the current user
my_submissions = as.list(Submission() |> synGetUserSubmissions(
  evaluation_id = "9999999"
))
print(sprintf("Found %d of my submissions", length(my_submissions)))

## Example 4: Delete a Submission.
library(synapser)
synLogin()

Submission(id = "9999999") |> synDelete()
print("Submission deleted successfully")

## End(Not run)

```

SubmissionBundle

*SubmissionBundle***Description**

A SubmissionBundle object represents a bundle containing a Synapse Submission and its accompanying SubmissionStatus. This bundle provides convenient access to both the submission data and its current status in a single object.

<https://rest-docs.synapse.org/rest/org/sagebionetworks/evaluation/model/SubmissionBundle.html>

Usage

```
SubmissionBundle(submission=NULL, submission_status=NULL)
```

Arguments

submission (Optional[Submission]) A Submission to a Synapse Evaluation is a pointer to a versioned Entity. Submissions are immutable, so we archive a copy of the EntityBundle at the time of submission.

submission_status (Optional[SubmissionStatus]) A SubmissionStatus is a secondary, mutable object associated with a Submission. This object should be used to contain scoring data about the Submission.

Format

An S3-style class object.

Value

NULL

Methods

- `SubmissionBundle(submission=NULL, submission_status=NULL)`: Constructor for [SubmissionBundle](#)
- `synGetEvaluationSubmissionBundles(cls, evaluation_id, status=NULL, synapse_client=NULL)`: Retrieves bundled Submissions and SubmissionStatuses for a given Evaluation.
- `synGetUserSubmissionBundles(cls, evaluation_id, synapse_client=NULL)`: Retrieves all user bundled Submissions and SubmissionStatuses for a specified Evaluation.

Examples

```
## Not run:
## Example 1: Retrieve submission bundles for an evaluation.
library(synapser)
synLogin()

# Get all submission bundles for an evaluation
bundles = SubmissionBundle() |> synGetEvaluationSubmissionBundles(
  evaluation_id = "9999999",
  status = "SCORED"
)

for (bundle in as.list(bundles)) {
  submission_id = if (is.null(bundle$submission)) "N/A" else bundle$submission$id
  status = if (is.null(bundle$submission_status)) "N/A" else bundle$submission_status$status
  print(sprintf("Submission ID: %s", submission_id))
  print(sprintf("Status: %s", status))
}

## Example 2: Retrieve user submission bundles for an evaluation.
library(synapser)
synLogin()

# Get current user's submission bundles for an evaluation
user_bundles = SubmissionBundle() |> synGetUserSubmissionBundles(
  evaluation_id = "9999999"
)

for (bundle in as.list(user_bundles)) {
  print(sprintf("Submission ID: %s", bundle$submission$id))
  print(sprintf("Status: %s", bundle$submission_status$status))
}

## End(Not run)
```


Description

A SubmissionStatus is a secondary, mutable object associated with a Submission. This object should be used to contain scoring data about the Submission. <https://rest-docs.synapse.org/rest/org/sagebionetworks/evaluation/model/SubmissionStatus.html>

Usage

```
SubmissionStatus(id=NULL, etag=NULL, modified_on=NULL, status=NULL, score=NULL, report=NULL, annotations=NULL)
```

Arguments

id	(Optional[character]) The unique, immutable Synapse ID of the Submission.
etag	(Optional[character]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. The eTag changes every time a SubmissionStatus is updated; it is used to detect when a client's copy of a SubmissionStatus is out-of-date.
modified_on	(Optional[character]) The date on which this SubmissionStatus was last modified.
status	(Optional[character]) The possible states of a Synapse Submission (e.g., RECEIVED, VALIDATED, SCORED).
score	(Optional[numeric]) This field is deprecated and should not be used. Use the 'submission_annotations' field instead.
report	(Optional[character]) This field is deprecated and should not be used. Use the 'submission_annotations' field instead.
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Primary container object for Annotations on a Synapse object. These annotations use the legacy format and do not show up in a submission view. The visibility is controlled by private_status_annotations.
submission_annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Submission Annotations are additional key-value pair metadata that are associated with an object. These annotations use the modern nested format and show up in a submission view.
private_status_annotations	(Optional[logical]) Indicates whether the annotations (not to be confused with submission annotations) are private (TRUE) or public (FALSE). Default is TRUE. This controls the visibility of the 'annotations' field.
entity_id	(Optional[character]) The Synapse ID of the Entity in this Submission.
version_number	(Optional[integer]) The version number of the Entity in this Submission.
status_version	(Optional[integer]) A version of the status, auto-generated and auto-incremented by the system and read-only to the client.
can_cancel	(Optional[logical]) Can this submission be cancelled? By default, this will be set to FALSE. Users can read this value. Only the queue's scoring application can change this value.

cancel_requested

(Optional[logical]) Has user requested to cancel this submission? By default, this will be set to FALSE. Submission owner can read and request to change this value.

Format

An S3-style class object.

Value

NULL

Methods

- `SubmissionStatus(id=NULL, etag=NULL, modified_on=NULL, status=NULL, score=NULL, report=NULL, annotations=list(), submission_annotations=list(), private_status_annotations=TRUE, entity_id=NULL, version_number=NULL, status_version=NULL, can_cancel=FALSE, cancel_requested=FALSE)`
Constructor for [SubmissionStatus](#)
- `synBatchUpdateSubmissionStatuses(evaluation_id, statuses, is_first_batch=TRUE, is_last_batch=TRUE, batch_token=NULL, synapse_client=NULL)`: Update multiple SubmissionStatuses. The maximum batch size is 500.
- `synGetAllSubmissionStatuses(evaluation_id, status=NULL, limit=10L, offset=0L, synapse_client=NULL)`: Gets a collection of SubmissionStatuses to a specified Evaluation.

Examples

```
## Not run:
## Example 1: Retrieve and update a SubmissionStatus.
# This example demonstrates the basic workflow of retrieving an existing submission status, updating its fields,
library(synapser)
synLogin()

# Get a submission status
status = SubmissionStatus(id = "9999999") |> synGet()

# Update the status
status$status = "SCORED"
status$submission_annotations = list(score = list(85.5), feedback = list("Good work!"))
status = status |> synStore()
print(status)

## Example 2: Get all submission statuses for an evaluation.
# Retrieve multiple submission statuses for an evaluation queue with optional filtering.
library(synapser)
synLogin()

# Get all RECEIVED statuses for an evaluation
statuses = synGetAllSubmissionStatuses(
  evaluation_id = "9999999",
  status = "RECEIVED",
```

```

        limit = 100L
    )

    print(sprintf("Found %d submission statuses", length(statuses)))
    for (status in statuses) {
        print(sprintf("Status ID: %s, Status: %s", status$id, status$status))
    }

## Example 3: Batch update multiple submission statuses.
# Efficiently update multiple submission statuses in a single operation.
library(synapser)
synLogin()

# Retrieve statuses to update
statuses = synGetAllSubmissionStatuses(
    evaluation_id = "9999999",
    status = "RECEIVED"
)

# Update each status
for (status in statuses) {
    status$status = "SCORED"
    status$submission_annotatons = list(
        validation_score = 95.0,
        comments = "Passed validation"
    )
}

# Batch update all statuses
response = synBatchUpdateSubmissionStatuses(
    evaluation_id = "9999999",
    statuses = statuses
)
print(response)

## End(Not run)

```

SubmissionView

Constructor for objects of type SubmissionView

Description

A SubmissionView object represents the metadata of a Synapse Submission View. <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/table/SubmissionView.html>>

Usage

```
SubmissionView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_ty
```

Arguments

id	(Optional[str]) The unique immutable ID for this entity. A new ID will be generated for new Entities. Once issued, this ID is guaranteed to never change or be re-issued.
name	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses.
parent_id	(Optional[str]) The ID of the Entity that is the parent of this Entity.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance.
version_number	(Optional[int]) The version number issued to this version on the object.
view_entity_type	(str) The API model string for the type of view. This is used to determine the default columns that are added to the table. Defaults to "submissionview".
view_type_mask	(Optional[int]) Bit mask representing the types to include in the view. This is used to determine the default columns that are added to the table. See https://rest-docs.synapse.org/rest/GET/column/tableview/defaults.html for the bit mask values.
include_default_columns	<p>(Optional[bool]) When creating a submission view, specifies if default columns should be included. Default columns are columns that are automatically added to the view. These columns are managed by Synapse and cannot be modified. If you attempt to create a column with the same name as a default column, you will receive a warning when you store the view.</p> <p>include_default_columns is only used if this is the first time that the view is being stored. If you are updating an existing view this attribute will be ignored.</p> <p>The column you are overriding will not behave the same as a default column. For example, suppose you create a column called id on a view. When using a default column, the id stores the Synapse ID of each of the entities included in the scope of the view. If you override the id column with a new column, the id column will no longer store the Synapse ID of the entities in the view. Instead, it will store the values you provide when you store the view. It will be stored as an annotation on the entity for the row you are modifying.</p>
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
created_on	(Optional[str]) The date this entity was created.
modified_on	(Optional[str]) The date this entity was last modified.
created_by	(Optional[str]) The ID of the user that created this entity.

<code>modified_by</code>	(Optional[str]) The ID of the user that last modified this entity.
<code>version_label</code>	(Optional[str]) The version label for this entity.
<code>version_comment</code>	(Optional[str]) The version comment for this entity.
<code>is_latest_version</code>	(Optional[bool]) If this is the latest version of the object.
<code>columns</code>	(Optional[Union[list[Column], OrderedDict[str, Column], dict[str, Column]]]) The columns of this submission view. This is an ordered named list where the key is the name of the column and the value is the Column object. When creating a new instance of a SubmissionView object you may pass any of the following types as the columns argument: - A list of Column objects - A named list where the key is the name of the column and the value is the Column object

The order of the columns will be the order they are stored in Synapse. If you need to reorder the columns the recommended approach is to use the [synReorderColumn\(\)](#) function. Additionally, you may add, and delete columns using the [synAddColumn\(\)](#), and [synDeleteColumn\(\)](#) functions on your view class instance.

You may modify the attributes of the Column object to change the column type, name, or other attributes. For example, suppose you'd like to change a column from a INTEGER to a DOUBLE. You can do so by changing the column type attribute of the Column object. The next time you store the view the column will be updated in Synapse with the new type.

```
library(synapser)
synLogin()
```

```
view = SubmissionView(id = "syn1234") |> synGet()
view$columns[["my_column"]][column_type = "DOUBLE"
view = view |> synStore()
```

Note that the keys in this named list should match the column names as they are in Synapse. However, know that the name attribute of the Column object is used for all interactions with the Synapse API. The named list key is purely for the usage of this interface. For example, if you wish to rename a column you may do so by changing the name attribute of the Column object. The key in the named list does not need to be changed. The next time you store the view the column will be updated in Synapse with the new name and the key in the named list will be updated.

<code>annotations</code>	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the submission view. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with
--------------------------	--

	all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.
<code>is_search_enabled</code>	(Optional[bool]) When creating or updating a table or view, specifies if full-text search should be enabled. Note that enabling full-text search might slow down the indexing of the table or view.
<code>scope_ids</code>	(list[str]) The list of container IDs that define the scope of this view. For submission views, this is the list of evaluation queues that the view is associated with.

Format

An S3-style class object.

Value

`NULL`

Methods

- `SubmissionView(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, view_entity_type="submissionview", view_type_mask=NULL, include_default_columns=TRUE, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, columns=list(), annotations=list(), is_search_enabled=NULL, scope_ids=list())`: Constructor for [SubmissionView](#)
- `synAddColumn(instance, column, index=NULL)`: Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this `SubmissionView` instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the `SubmissionView` instance. `$columns` is a named list where the key is the name of the column and the value is the `Column` object.

- `synDeleteColumn(instance, name)`: Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this `SubmissionView` instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.
- `synReorderColumn(instance, name, index)`: Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this `SubmissionView` instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a `DataFrame` or downloading the results to a file.

- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API:

<<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

- `synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_timeout=120L, synapse_client=NULL)`: Creates a snapshot of the SubmissionView entity. Synapse handles snapshot creation differently for Table- and View-like entities. View snapshots are created using the asynchronous job API.

Making a snapshot of a view allows you to create an immutable version of the view at the time of the snapshot. This is useful to create checkpoints in time that you may go back and reference, or use in a publication. Snapshots are immutable and cannot be changed. They may only be deleted.

Examples

```
## Not run:
## Create a new SubmissionView.
library(synapser)
synLogin()

my_submission_view = SubmissionView(
  name = "My Submission View",
  parent_id = "syn1234",
  scope_ids = list("syn5678", "syn6789") # IDs of evaluation queues
) |> synStore()
my_submission_view

## End(Not run)
```

synAddColumn

Dataset : add_column

Description

Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this Dataset instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the Dataset instance. `$columns` is a named list where the key is the name of the column and the value is the Column object.

Usage

```
synAddColumn(instance, column, index=NULL)
```


Arguments

instance	(Dataset) The Dataset instance to operate on.
column	(Column or list of Column) The column(s) to add, may be a single Column object or a list of Column objects.
index	(integer) The index to insert the column at. If not passed in the column will be added to the end of the list.

Value

NULL

Examples

```
## Not run:
## Example 1: Adding a single column
# This example shows how you may add a single column to a dataset and then
# store the change back in Synapse.
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

dataset |> synAddColumn(
  Column(name = "my_column", column_type = "STRING")
)
dataset |> synStore()

## Example 2: Adding multiple columns
# This example shows how you may add multiple columns to a dataset and then
# store the change back in Synapse.
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

dataset |> synAddColumn(list(
  Column(name = "my_column", column_type = "STRING"),
  Column(name = "my_column2", column_type = "INTEGER")
))
dataset |> synStore()

## Example 3: Adding a column at a specific index
# This example shows how you may add a column at a specific index to a
# dataset and then store the change back in Synapse. If the index is out of
# bounds the column will be added to the end of the list.
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

dataset |> synAddColumn(
```

```

      Column(name = "my_column", column_type = "STRING"),
      # Add the column at the beginning of the list
      index = 0L
    )
    dataset |> synStore()

## End(Not run)

```

synAddColumn

DatasetCollection : add_column

Description

Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this `DatasetCollection` instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the `DatasetCollection` instance. `$columns` is a named list where the key is the name of the column and the value is the `Column` object.

Usage

```
synAddColumn(instance, column, index=NULL)
```

Arguments

instance	(<code>DatasetCollection</code>) The <code>DatasetCollection</code> instance to operate on.
column	(<code>Column</code> or list of <code>Column</code>) The column(s) to add, may be a single <code>Column</code> object or a list of <code>Column</code> objects.
index	(integer) The index to insert the column at. If not passed in the column will be added to the end of the list.

Value

NULL

Examples

```

## Not run:
## Example 1: Adding a single column
# This example shows how you may add a single column to a dataset collection
# and then store the change back in Synapse.
library(synapser)
synLogin()

collection = DatasetCollection(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

collection |> synAddColumn(
  Column(name = "my_column", column_type = "STRING")
)

```

```

    )
    collection |> synStore()

## Example 2: Adding multiple columns
# This example shows how you may add multiple columns to a dataset collection
# and then store the change back in Synapse.

library(synapser)
synLogin()

collection = DatasetCollection(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

collection |> synAddColumn(list(
  Column(name = "my_column", column_type = "STRING"),
  Column(name = "my_column2", column_type = "INTEGER")
))
collection |> synStore()

## Example 3: Adding a column at a specific index
# This example shows how you may add a column at a specific index to a
# dataset collection and then store the change back in Synapse. If the index
# is out of bounds the column will be added to the end of the list.

library(synapser)
synLogin()

collection = DatasetCollection(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

collection |> synAddColumn(
  Column(name = "my_column", column_type = "STRING"),
  # Add the column at the beginning of the list
  index = 0L
)
collection |> synStore()

## End(Not run)

```

synAddColumn

*EntityView : add_column***Description**

Add column(s) to the EntityView. Note that this does not store the column(s) in Synapse. You must call the [synStore\(\)](#) function on this EntityView instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the \$columns list.

This function will add an item to the \$columns attribute of the EntityView instance. \$columns is a named list where the key is the name of the column and the value is the Column object.

Usage

```
synAddColumn(instance, column, index=NULL)
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
column	(Column or list of Column) The column(s) to add, may be a single Column object or a list of Column objects.
index	(int) The index to insert the column at. If not passed in the column will be added to the end of the list.

Value

NULL

Examples

```
## Not run:
## Example 1: Adding a single column
# This example shows how you may add a single column to an EntityView and then store
# the change back in Synapse.

library(synapser)
synLogin()

entityview = EntityView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

entityview |> synAddColumn(
  Column(name = "my_column", column_type = "STRING")
)
entityview |> synStore()

## Example 2: Adding multiple columns
# This example shows how you may add multiple columns to an EntityView and then store
# the change back in Synapse.

library(synapser)
synLogin()

entityview = EntityView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

entityview |> synAddColumn(list(
  Column(name = "my_column", column_type = "STRING"),
  Column(name = "my_column2", column_type = "INTEGER")
))
entityview |> synStore()

## Example 3: Adding a column at a specific index
# This example shows how you may add a column at a specific index to an EntityView
# and then store the change back in Synapse. If the index is out of bounds the
# column will be added to the end of the list.

library(synapser)
synLogin()
```

```

entityview = EntityView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

entityview |> synAddColumn(
  Column(name = "my_column", column_type = "STRING"),
  # Add the column at the beginning of the list
  index = 0L
)
entityview |> synStore()

## End(Not run)

```

synAddColumn

*SubmissionView : add_column***Description**

Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the [synStore\(\)](#) function on this SubmissionView instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the \$columns list.

This function will add an item to the \$columns attribute of the SubmissionView instance. \$columns is a named list where the key is the name of the column and the value is the Column object.

Usage

```
synAddColumn(instance, column, index=NULL)
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
column	(Column or list of Column) The column(s) to add, may be a single Column object or a list of Column objects.
index	(int) The index to insert the column at. If not passed in the column will be added to the end of the list.

Value

NULL

Examples

```

## Not run:
## Example 1: Adding a single column
# This example shows how you may add a single column to a submission view and
# then store the change back in Synapse.
library(synapser)
synLogin()

view = SubmissionView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

```

```

view |> synAddColumn(
  Column(name = "my_column", column_type = "STRING")
)
view |> synStore()

## Example 2: Adding multiple columns
# This example shows how you may add multiple columns to a submission view and
# then store the change back in Synapse.

library(synapser)
synLogin()

view = SubmissionView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

view |> synAddColumn(list(
  Column(name = "my_column2", column_type = "STRING"),
  Column(name = "my_column3", column_type = "INTEGER")
))
view |> synStore()

## Example 3: Adding a column at a specific index
# This example shows how you may add a column at a specific index to a
# submission view and then store the change back in Synapse. If the index
# is out of bounds the column will be added to the end of the list.

library(synapser)
synLogin()

view = SubmissionView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

view |> synAddColumn(
  Column(name = "my_column4", column_type = "STRING"),
  # Add the column at the beginning of the list
  index = 0L
)
view |> synStore()

## End(Not run)

```

synAddColumn

Table : *add_column*

Description

Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the [synStore\(\)](#) function on this Table instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the \$columns list.

This function will add an item to the \$columns attribute of the Table instance. \$columns is a named list where the key is the name of the column and the value is the Column object.

Usage

```
synAddColumn(instance, column, index=NULL)
```

Arguments

instance	(Table) The Table instance to operate on.
column	(Column or list of Column) The column(s) to add, may be a single Column object or a list of Column objects.
index	(int) The index to insert the column at. If not passed in the column will be added to the end of the list.

Value

NULL

Examples

```
## Not run:
## Example 1: Adding a single column
# This example shows how you may add a single column to a table and then store
# the change back in Synapse.
library(synapser)
synLogin()

table = Table(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

table |> synAddColumn(
  Column(name = "my_column", column_type = "STRING")
)
table |> synStore()

## Example 2: Adding multiple columns
# This example shows how you may add multiple columns to a table and then store
# the change back in Synapse.

library(synapser)
synLogin()

table = Table(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

table |> synAddColumn(list(
  Column(name = "my_column2", column_type = "STRING"),
  Column(name = "my_column3", column_type = "INTEGER")
))
table |> synStore()

## Example 3: Adding a column at a specific index
# This example shows how you may add a column at a specific index to a table
# and then store the change back in Synapse. If the index is out of bounds the
# column will be added to the end of the list.
```

```

library(synapser)
synLogin()

table = Table(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

table |> synAddColumn(
  Column(name = "my_column4", column_type = "STRING"),
  # Add the column at the beginning of the list
  index = 0L
)
table |> synStore()

## End(Not run)

```

synAddItem

Dataset : add_item

Description

Adds an item in the form of an [EntityRef](#) to the dataset. For Folders, children are added recursively. Effect is not seen until the dataset is stored.

Usage

```
synAddItem(instance, item, synapse_client=NULL)
```

Arguments

<code>instance</code>	(Dataset) The Dataset instance to operate on.
<code>item</code>	(EntityRef, File, or Folder) Entity to add to the dataset. Must be an EntityRef , File, or Folder.
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

Raises an error if the item is not an EntityRef, File, or Folder.

Examples

```
## Not run:
## Example 1: Add a file to a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synAddItem(File(id = "syn1235"))
my_dataset |> synStore()

## Example 2: Add a folder to a dataset.
# All child files are recursively added to the dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synAddItem(Folder(id = "syn1236"))
my_dataset |> synStore()

## Example 3: Add an entity reference to a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synAddItem(EntityRef(id = "syn1237", version = 1L))
my_dataset |> synStore()

## End(Not run)
```

synAddItem	<i>DatasetCollection : add_item</i>
------------	-------------------------------------

Description

Adds a dataset to the dataset collection. Effect is not seen until the dataset collection is stored.

Usage

```
synAddItem(instance, item)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
item	(Dataset or EntityRef) Dataset to add to the collection. Must be a Dataset or an EntityRef pointing to one.

Value

NULL

Errors

Raises an error if the item is not a Dataset or EntityRef.

Examples

```
## Not run:
## Add a Dataset to a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synAddItem(Dataset(id = "syn1235"))
my_collection |> synStore()

## End(Not run)
```

synBatchUpdateSubmissionStatuses

SubmissionStatus : batch_update_submission_statuses

Description

Update multiple SubmissionStatuses. The maximum batch size is 500.

Usage

```
synBatchUpdateSubmissionStatuses(evaluation_id, statuses, is_first_batch=TRUE, is_last_batch=TRUE, batch_token, synapse_client)
```

Arguments

evaluation_id	(character) The ID of the Evaluation to which the SubmissionStatus objects belong.
statuses	(list[SubmissionStatus]) List of SubmissionStatus objects to update.
is_first_batch	(logical) Whether this is the first batch in the series. Default TRUE.
is_last_batch	(logical) Whether this is the last batch in the series. Default TRUE.
batch_token	(Optional[character]) Token from previous batch response (required for all but first batch).
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A BatchUploadResponse as a named list containing the batch token and other response information.

Examples

```
## Not run:
## Batch update submission statuses
# Update multiple submission statuses in a single batch operation for efficiency.
library(synapser)
synLogin()

# Retrieve existing statuses to update
statuses = synGetAllSubmissionStatuses(
  evaluation_id = "9999999",
  status = "RECEIVED"
)

# Modify statuses as needed
for (status in statuses) {
  status$status = "SCORED"
}

# Update statuses in batch
response = synBatchUpdateSubmissionStatuses(
  evaluation_id = "9999999",
  statuses = statuses,
  is_first_batch = TRUE,
  is_last_batch = TRUE
)
print(response)

## End(Not run)
```

synBindSchema

EntityView : bind_schema

Description

Bind a JSON schema to the entity.

Usage

```
synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)
```

Arguments

instance (EntityView) The EntityView instance to operate on.

json_schema_uri (str) The URI of the JSON schema to bind to the entity.

enable_derived_annotations (bool) If TRUE, enable derived annotations. Defaults to FALSE.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the JSON schema binding.

Examples

```
## Not run:
## Using this function
# Binding JSON schema to an EntityView. This example expects that you
# have a Synapse project to use. Set the PROJECT_NAME variable to
# your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VIEW_NAME = "test_script_entityview"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test EntityView of the files in the project
test_view = EntityView(
  name = VIEW_NAME,
  parent_id = PROJECT_ID,
  scope_ids = list(PROJECT_ID),
  view_type_mask = 1L # 1 = files
) |> synStore()
cat("Created test EntityView:", VIEW_NAME, "\n")

# Bind JSON schema to the EntityView
bound_schema = test_view |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
```

```

        enable_derived_annotations = TRUE
    )
    cat("Result from binding schema to EntityView:\n")
    bound_schema

## End(Not run)

```

synBindSchema

File : bind_schema

Description

Bind a JSON schema to the entity.

Usage

```
synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)
```

Arguments

instance (File) The File instance to operate on.

json_schema_uri (str) The URI of the JSON schema to bind to the entity.

enable_derived_annotations (bool) If TRUE, enable derived annotations. Defaults to FALSE.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the JSON schema binding.

Examples

```

## Not run:
## Using this function
# Binding JSON schema to a folder or a file. This example expects that you
# have a Synapse project to use, and a file to upload. Set the PROJECT_NAME
# and FILE_PATH variables to your project name and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name

```

```

SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to file:\n")
bound_schema_file

## End(Not run)

```

synBindSchema

Folder : bind_schema

Description

Bind a JSON schema to the entity.

Usage

```
synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)
```

Arguments

`instance` (Folder) The Folder instance to operate on.

`json_schema_uri` (str) The URI of the JSON schema to bind to the entity.

`enable_derived_annotations` (bool) If TRUE, enable derived annotations. Defaults to FALSE.

`synapse_client` (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the JSON schema binding.

Examples

```
## Not run:
## Using this function
# Binding JSON schema to a folder or a file. This example expects that you
# have a Synapse project to use, and a file to upload. Set the PROJECT_NAME
# and FILE_PATH variables to your project name and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
```

```

test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to file:\n")
bound_schema_file

## End(Not run)

```

synBindSchema

Project : bind_schema

Description

Bind a JSON schema to the entity.

Usage

```
synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

json_schema_uri (str) The URI of the JSON schema to bind to the entity.

enable_derived_annotations (bool) If TRUE, enable derived annotations. Defaults to FALSE.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the JSON schema binding.

Examples

```
## Not run:
## Using this function
# Binding JSON schema to a project. This example expects that you
# have a Synapse project to use. Set the PROJECT_NAME variable to
# your project name.
library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to project:\n")
bound_schema

## End(Not run)
```

synBindSchema	<i>Table : bind_schema</i>
---------------	----------------------------

Description

Bind a JSON schema to the entity.

Usage

```
synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)
```

Arguments

`instance` (Table) The Table instance to operate on.

`json_schema_uri` (str) The URI of the JSON schema to bind to the entity.

`enable_derived_annotations` (bool) If TRUE, enable derived annotations. Defaults to FALSE.

`synapse_client` (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the JSON schema binding.

Examples

```
## Not run:
## Using this function
# Binding JSON schema to a table. This example expects that you
# have a Synapse project to use. Set the PROJECT_NAME variable to
# your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
TABLE_NAME = "test_script_table"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()
```

```

    schema_definition = list(
      "$id" = "mySchema",
      type = "object",
      properties = list(
        foo = list(type = "string"),
        bar = list(type = "integer")
      ),
      required = list("foo")
    )
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test table
test_table = Table(name = TABLE_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test table:", TABLE_NAME, "\n")

# Bind JSON schema to the table
bound_schema = test_table |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Result from binding schema to table:\n")
bound_schema

## End(Not run)

```

synCancel

Submission : cancel

Description

Cancel a Submission. Only the user who created the Submission may cancel it.

Usage

```
synCancel(instance, synapse_client=NULL)
```

Arguments

instance (Submission) The Submission instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The updated Submission object.

Errors

Raises an error if the submission does not have an ID to cancel.

Examples

```
## Not run:
## Cancel a submission
library(synapser)

synLogin()

submission = Submission(id = "syn1234")
canceled_submission = submission |> synCancel()
print(sprintf("Canceled submission: %s", canceled_submission$id))

## End(Not run)
```

synChangeMetadata	<i>File : change_metadata</i>
-------------------	-------------------------------

Description

Change File Entity metadata for properties that are immutable after creation through the store method.

Usage

```
synChangeMetadata(instance, name=NULL, download_as=NULL, content_type=NULL, synapse_client=NULL)
```

Arguments

instance	(File) The File instance to operate on.
name	(Optional[str]) Specify to change the filename of a file as seen on Synapse.
download_as	(Optional[str]) Specify filename to change the filename of a filehandle.
content_type	(Optional[str]) Specify content type to change the content type of a filehandle.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The file object.

Errors

An error is raised if the file does not have an ID to change metadata.

Examples

```
## Not run:
## Using this function
# Can be used to change the filename, the filename when the file is
# downloaded, or the file content-type without downloading:
file_entity = File(id = "syn123", download_file = FALSE) |> synGet()
cat(basename(file_entity$path), "\n") # prints, e.g., "my_file.txt"

file_entity = file_entity |> synChangeMetadata(
  name = "my_new_name_file.txt",
  download_as = "my_new_downloadAs_name_file.txt",
  content_type = "text/plain"
)
cat(basename(file_entity$path), "\n") # prints, "my_new_downloadAs_name_file.txt"
cat(file_entity$name, "\n") # prints, "my_new_name_file.txt"

## End(Not run)
```

synCopy

File : copy

Description

Copy the file to another Synapse location. Defaults to the latest version of the file, or the version_number specified in the instance.

Usage

```
synCopy(instance, parent_id, update_existing=FALSE, copy_annotations=TRUE, copy_activity="traceback",
```

Arguments

instance	(File) The File instance to operate on.
parent_id	(str) Synapse ID of a folder/project that the copied entity is being copied to
update_existing	(bool) When the destination has a file that has the same name, users can choose to update that file.
copy_annotations	(bool) TRUE to copy the annotations.
copy_activity	(Optional[str]) Has three options to set the activity of the copied file: - traceback: Creates a copy of the source files Activity. - existing: Link to the source file's original Activity (if it exists) - NULL: No activity is set
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The copied file object.

Errors

An error is raised if the file does not have an ID and parent_id to copy.

Examples

```
## Not run:
## Using this function
# Assuming you have a file with the ID "syn123" and you want to copy it to a
# folder with the ID "syn456":
new_file_instance = File(id = "syn123") |> synCopy(parent_id = "syn456")

# Copy the file but do not persist annotations or activity:
new_file_instance = File(id = "syn123") |> synCopy(parent_id = "syn456", copy_annotations = FALSE, copy_activity = FALSE)

## End(Not run)
```

synCopy	<i>Folder : copy</i>
---------	----------------------

Description

Copy the folder to another Synapse location. This will recursively copy all Tables, Links, Files, and Folders within the folder.

Usage

```
synCopy(instance, parent_id, copy_annotations=TRUE, exclude_types=NULL, file_update_existing=FALSE, file_copy_activity=TRUE)
```

Arguments

- instance (Folder) The Folder instance to operate on.
- parent_id (str) Synapse ID of a folder/project that the copied entity is being copied to
- copy_annotations (bool) TRUE to copy the annotations.
- exclude_types (Optional[list[str]]) A list of entity types list("file", "table", "link") which determines which entity types to not copy. Defaults to an empty list.
- file_update_existing (bool) When the destination has a file that has the same name, users can choose to update that file.
- file_copy_activity (Optional[str]) Has three options to set the activity of the copied file:
 - traceback: Creates a copy of the source files Activity.
 - existing: Link to the source file's original Activity (if it exists)
 - NULL: No activity is set

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The copied folder object.

Errors

An error is raised if the folder does not have an ID and `parent_id` to copy.

Examples

```
## Not run:
## Using this function
# Assuming you have a folder with the ID "syn123" and you want to copy it to a
# project with the ID "syn456":
new_folder_instance = Folder(id = "syn123") |> synCopy(parent_id = "syn456")

# Copy the folder but do not persist annotations:
new_folder_instance = Folder(id = "syn123") |> synCopy(parent_id = "syn456", copy_annotations = FALSE)

## End(Not run)
```

synCopy

Project : copy

Description

You must have already created the Project you will be copying to. It will have its own Synapse ID and unique name that you will use as the `destination_id`.

Copy the project to another Synapse project. This will recursively copy all Tables, Links, Files, and Folders within the project.

Usage

```
synCopy(instance, destination_id, copy_annotations=TRUE, copy_wiki=TRUE, exclude_types=NULL, file_upd
```

Arguments

`instance` (Project) The Project instance to operate on.

`destination_id` (str) Synapse ID of a project to copy to.

`copy_annotations` (bool) TRUE to copy the annotations.

`copy_wiki` (bool) TRUE to copy the wiki pages.

`exclude_types` (Optional[list[str]]) A list of entity types list("file", "table", "link") which determines which entity types to not copy. Defaults to an empty list.

`file_update_existing` (bool) When the destination has a file that has the same name, users can choose to update that file.

`file_copy_activity` (Optional[str]) Has three options to set the activity of the copied file:

- `traceback`: Creates a copy of the source files Activity.
- `existing`: Link to the source file's original Activity (if it exists)
- `NULL`: No activity is set

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The copied project object.

Errors

An error is raised if the project does not have an ID and `destination_id` to copy.

Examples

```
## Not run:
## Using this function
# Assuming you have a project with the ID "syn123" and you want to copy it to a
# project with the ID "syn456":
library(synapser)
synLogin()

new_instance = Project(id = "syn123") |> synCopy(destination_id = "syn456")

# Copy the project but do not persist annotations:
new_instance = Project(id = "syn123") |> synCopy(destination_id = "syn456", copy_annotations = FALSE)

## End(Not run)
```

synCopy

WikiPage : copy

Description

Copy the wiki page tree of the owner entity to another entity and update internal links.

If `id` is set on this `WikiPage`, only the sub-tree rooted at that wiki page is copied. Otherwise the entire wiki of the owner entity is copied.

Usage

```
synCopy(instance, destination_owner_id, destination_sub_page_id=NULL, update_links=TRUE, entity_map=)
```


Arguments

instance	(WikiPage) The WikiPage instance to operate on.
destination_owner_id	(character) The Synapse ID of the entity that the wiki will be copied to.
destination_sub_page_id	(Optional[character]) Optional ID of a wiki page that already exists in the destination. The root of the copied tree is written into that page, replacing its title, markdown, and attachments, and the rest of the copied pages are created beneath it. Required when the destination entity already has a root wiki page.
update_links	(logical) Update all the internal links so that they point at the copied wiki pages. For example, syn1234/wiki/34345 becomes syn3345/wiki/49508. Defaults to TRUE.
entity_map	(Optional[named list]) A mapping of old Synapse IDs to new Synapse IDs, for example list(syn1234 = "syn2345"). If provided, the Synapse IDs referenced in the markdown of the copied wiki pages are updated, for example syn1234 becomes syn2345. If omitted, Synapse IDs are left unchanged.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of [WikiHeader](#) objects for the destination entity.

Errors

Raises an error if owner_id is not provided or is not a Synapse ID, if destination_owner_id is not a Synapse ID, if a key or value of entity_map is not a Synapse ID, if id is set but no wiki page with that ID exists in the owner entity, if destination_sub_page_id does not exist in the destination entity, if the destination entity already has a root wiki page and destination_sub_page_id is not provided, or if copying an attachment file handle fails.

Examples

```
## Not run:
## Example 1: Copy the entire wiki of an entity to another entity
# This example shows how to copy all wiki pages from one project to another.
library(synapser)
synLogin()

new_wiki_headers = WikiPage(owner_id = "syn123") |> synCopy(
  destination_owner_id = "syn456"
)
print(new_wiki_headers)

## Example 2: Copy a wiki sub-tree and update Synapse ID references
# This example shows how to copy a specific wiki page and its sub-pages,
# rewriting references to syn1234 so they point at syn2345.
new_wiki_headers = WikiPage(owner_id = "syn123", id = "34345") |> synCopy(
  destination_owner_id = "syn456",
```

```

        entity_map = list(syn1234 = "syn2345")
    )
    print(new_wiki_headers)

## End(Not run)

```

synCreate

Team : create

Description

Creates a new team on Synapse.

Usage

```
synCreate(instance, synapse_client=NULL)
```

Arguments

instance (Team) The Team instance to operate on.

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

Team: The Team object.

Examples

```

## Not run:
## Create a new team
library(synapser)
synLogin()

team = Team(
  name = "My Uniquely Named Team",
  description = "A team for my project collaborators",
  can_public_join = FALSE
) |> synCreate()

## End(Not run)

```

synDelete

*synDelete***Description**

Factory method to delete any Synapse entity.

This method serves as a unified interface for deleting any type of Synapse entity. It automatically applies the appropriate delete logic based on the entity type.

Version Deletion Behavior: The function supports deleting specific versions of entities with the following precedence order:

1. **version parameter** (highest priority): When provided, this version number will be used for deletion, overriding any version information from the entity object or the entity ID string.
2. **Entity's version_number attribute** (secondary): If the entity object has a version_number attribute set and no explicit version parameter is provided, this version will be used.
3. **version_only parameter**: When set to TRUE, indicates that only a specific version should be deleted (not the entire entity). This parameter is only meaningful when combined with a version number from either the version parameter or the entity's version_number attribute.

Supported for version-specific deletion:

- String ID with version (e.g., "syn123.4")
- File, RecordSet (use version_only=TRUE)
- Table, Dataset, DatasetCollection, EntityView, MaterializedView, SubmissionView, VirtualTable (use version_only=TRUE)

Not supported for version-specific deletion:

- Project, Folder, Evaluation, Team, Organization, CurationTask, Grid, SearchIndex

Usage

```
synDelete(entity, version=NULL, version_only=FALSE, synapse_client=NULL)
```

Arguments

entity	(Union[str, CurationTask, Dataset, DatasetCollection, EntityView, Evaluation, File, Folder, Grid, JSONSchema, MaterializedView, Project, RecordSet, Organization, SearchIndex, SubmissionView, Table, Team, VirtualTable, DockerRepository]) The entity instance to delete, or a Synapse ID string (e.g., "syn123456" or "syn123456.4" for a specific version). Must be one of the supported entity types or a valid Synapse ID.
version	(Optional[Union[str, int]]) Optional version number to delete. Takes precedence over any version information in the entity object or ID string. When provided with version_only = TRUE, deletes only this specific version.
version_only	(bool) If TRUE, only the specified version will be deleted, not the entire entity. Requires a version number from either the version parameter or the entity's version_number attribute. This parameter is applicable for entities that support version-specific deletion (File, RecordSet, Table, Dataset, DatasetCollection, EntityView, MaterializedView, SubmissionView, VirtualTable, JSONSchema).

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity is not a supported type or not a valid Synapse ID.
- An error is raised if `version_only` is `TRUE` but no version number is available.

Examples

```
## Not run:
## Example 1: Deleting a file by object
# Delete a file from Synapse:
library(synapser)
synLogin()

file = File(id = "syn123456")
file |> synDelete()
print("File deleted successfully")

## Example 2: Deleting a specific version of a file
# Delete only version 2 of a file, keeping other versions:
library(synapser)
synLogin()

# Method 1: Using version parameter (highest priority)
file = File(id = "syn123456")
file |> synDelete(version = 2L, version_only = TRUE)

# Method 2: Using entity's version_number attribute
file = File(id = "syn123456", version_number = 2L)
file |> synDelete(version_only = TRUE)

# Method 3: Using synapse ID with version
synDelete("syn123456.2", version_only = TRUE)
print("File version 2 deleted successfully")

## Example 3: Deleting a file by ID string
# Delete a file from Synapse using just its ID:
library(synapser)
synLogin()

synDelete("syn123456")
print("Entity deleted successfully")

## Example 4: Deleting a project
# Delete a project from Synapse:
library(synapser)
```

```

synLogin()

project = Project(id = "syn123456")
project |> synDelete()
print("Project deleted successfully")

## Example 5: Deleting a table
# Delete a table from Synapse:
library(synapser)
synLogin()

table = Table(id = "syn123456")
table |> synDelete()
print("Table deleted successfully")

## Example 6: Deleting a team
# Delete a team from Synapse:
library(synapser)
synLogin()

team = Team(id = "123456")
team |> synDelete()
print("Team deleted successfully")

## End(Not run)

```

synDeleteColumn

Dataset : delete_column

Description

Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the [synStore\(\)](#) function on this Dataset instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.

Usage

```
synDeleteColumn(instance, name)
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
name	(character) The name of the column to delete.

Value

NULL

Examples

```
## Not run:
## Deleting a column
# This example shows how you may delete a column from a dataset and then
# store the change back in Synapse.
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

dataset |> synDeleteColumn(name = "my_column")
dataset |> synStore()

## End(Not run)
```

synDeleteColumn	<i>DatasetCollection : delete_column</i>
-----------------	--

Description

Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this `DatasetCollection` instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.

Usage

```
synDeleteColumn(instance, name)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
name	(character) The name of the column to delete.

Value

NULL

Examples

```
## Not run:
## Deleting a column
# This example shows how you may delete a column from a dataset collection
# and then store the change back in Synapse.
library(synapser)
synLogin()

collection = DatasetCollection(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))
```

```

collection |> synDeleteColumn(name = "my_column")
collection |> synStore()

## End(Not run)

```

synDeleteColumn	<i>EntityView : delete_column</i>
-----------------	-----------------------------------

Description

Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this `EntityView` instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.

Usage

```
synDeleteColumn(instance, name)
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
name	(str) The name of the column to delete.

Value

NULL

Examples

```

## Not run:
## Deleting a column
# This example shows how you may delete a column from an EntityView and then store
# the change back in Synapse.

library(synapser)
synLogin()

entityview = EntityView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

entityview |> synDeleteColumn(name = "my_column")
entityview |> synStore()

## End(Not run)

```

synDeleteColumn	<i>SubmissionView : delete_column</i>
-----------------	---------------------------------------

Description

Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the [synStore\(\)](#) function on this SubmissionView instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.

Usage

```
synDeleteColumn(instance, name)
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
name	(str) The name of the column to delete.

Value

NULL

Examples

```
## Not run:
## Deleting a column
# This example shows how you may delete a column from a submission view and
# then store the change back in Synapse.
library(synapser)
synLogin()

view = SubmissionView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

view |> synDeleteColumn(name = "my_column")
view |> synStore()

## End(Not run)
```

synDeleteColumn	<i>Table : delete_column</i>
-----------------	------------------------------

Description

Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the [synStore\(\)](#) function on this Table instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.

Usage

```
synDeleteColumn(instance, name)
```

Arguments

instance	(Table) The Table instance to operate on.
name	(str) The name of the column to delete.

Value

NULL

Examples

```
## Not run:
## Deleting a column
# This example shows how you may delete a column from a table and then store
# the change back in Synapse.
library(synapser)
synLogin()

table = Table(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

table |> synDeleteColumn(name = "my_column")
table |> synStore()

## End(Not run)
```

synDeleteInvitation *Team : delete_invitation*

Description

Deletes an open invitation to a team. Note: The client must be an administrator of the Team referenced by the invitation, or the invitee, to make this request.

Usage

```
synDeleteInvitation(invitation_id, synapse_client=NULL)
```

Arguments

invitation_id (character) The ID of the invitation to delete. This can be found on the invitations returned by [synGetOpenInvitations\(\)](#).

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Delete an open invitation to a team
# Cancel a pending invitation to a team.
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
open_invitations = team |> synGetOpenInvitations()
if (length(open_invitations) == 0) {
  print("No open invitations to delete.")
} else {
  # Delete the first open invitation
  synDeleteInvitation(invitation_id = open_invitations[[1]][["id"]])
}

## End(Not run)
```

 synDeletePermissions *Dataset : delete_permissions*

Description

Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permissions are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
include_self	(logical) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
include_container_content	(logical) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
recursive	(logical) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the synSyncFromSynapse() function.
target_entity_types	(Optional[list[character]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submission-view" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if <code>include_self=TRUE</code> .

<code>dry_run</code>	(logical) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
<code>show_acl_details</code>	(logical) When <code>dry_run=TRUE</code> , controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when <code>dry_run=FALSE</code> .
<code>show_files_in_containers</code>	(logical) When <code>dry_run=TRUE</code> , controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when <code>dry_run=FALSE</code> .
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted `CHANGE_PERMISSIONS` access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

Dataset(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
```

```

        include_container_content = TRUE
    )

    # Delete permissions for all items in the entire hierarchy (folders and their files)
    # Both recursive and include_container_content must be TRUE
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE
    )

    # Delete permissions only for folder entities within this folder recursively
    # and their contents
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("folder")
    )

    # Delete permissions only for files within this folder and all subfolders
    Folder(id = "syn123") |> synDeletePermissions(
        include_self = FALSE,
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("file")
    )

    # Delete permissions for specific entity types (e.g., tables and views)
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("table", "entityview", "materializedview")
    )

    # Dry run example: Log what would be deleted without making changes
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        dry_run = TRUE
    )

    ## End(Not run)

```

synDeletePermissions *DatasetCollection : delete_permissions*

Description

Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the

Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

<code>instance</code>	(DatasetCollection) The DatasetCollection instance to operate on.
<code>include_self</code>	(logical) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
<code>include_container_content</code>	(logical) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
<code>recursive</code>	(logical) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the <code>synSyncFromSynapse()</code> function.
<code>target_entity_types</code>	(Optional[list[character]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submission-view" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if <code>include_self=TRUE</code> .
<code>dry_run</code>	(logical) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
<code>show_acl_details</code>	(logical) When <code>dry_run=TRUE</code> , controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when <code>dry_run=FALSE</code> .

`show_files_in_containers` (logical) When `dry_run=TRUE`, controls whether files within containers are displayed in the preview. If `TRUE` (default), shows all files. If `FALSE`, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when `dry_run=FALSE`.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted `CHANGE_PERMISSIONS` access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

DatasetCollection(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)
```

```

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)

```

synDeletePermissions *EntityView : delete_permissions*

Description

Delete the entire Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
include_self	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
include_container_content	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if include_self=TRUE.
dry_run	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
show_acl_details	(bool) When dry_run=TRUE, controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.
show_files_in_containers	(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

EntityView(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
```

```

        target_entity_types = list("file")
    )

    # Delete permissions for specific entity types (e.g., tables and views)
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("table", "entityview", "materializedview")
    )

    # Dry run example: Log what would be deleted without making changes
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        dry_run = TRUE
    )

## End(Not run)

```

synDeletePermissions *File : delete_permissions*

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

instance (File) The File instance to operate on.

<code>include_self</code>	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
<code>include_container_content</code>	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
<code>recursive</code>	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the <code>synSyncFromSynapse()</code> function.
<code>target_entity_types</code>	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if <code>include_self=TRUE</code> .
<code>dry_run</code>	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
<code>show_acl_details</code>	(bool) When <code>dry_run=TRUE</code> , controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when <code>dry_run=FALSE</code> .
<code>show_files_in_containers</code>	(bool) When <code>dry_run=TRUE</code> , controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when <code>dry_run=FALSE</code> .
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted `CHANGE_PERMISSIONS` access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

File(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)
```

```
# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)
```

synDeletePermissions *Folder : delete_permissions*

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

<code>instance</code>	(Folder) The Folder instance to operate on.
<code>include_self</code>	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
<code>include_container_content</code>	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
<code>recursive</code>	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the synSyncFromSynapse() function.

target_entity_types	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if include_self=TRUE.
dry_run	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
show_acl_details	(bool) When dry_run=TRUE, controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.
show_files_in_containers	(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

Folder(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
```

```

Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)

```

synDeletePermissions *MaterializedView : delete_permissions*

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

<code>instance</code>	(MaterializedView) The MaterializedView instance to operate on.
<code>include_self</code>	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
<code>include_container_content</code>	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
<code>recursive</code>	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the synSyncFromSynapse() function.
<code>target_entity_types</code>	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if <code>include_self=TRUE</code> .
<code>dry_run</code>	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
<code>show_acl_details</code>	(bool) When <code>dry_run=TRUE</code> , controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default),

shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.

show_files_in_containers

(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

MaterializedView(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself and not files within its subfolder
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
```

```

        include_container_content = TRUE
    )

    # Delete permissions only for folder entities within this folder recursively
    # and their contents
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("folder")
    )

    # Delete permissions only for files within this folder and all subfolders
    Folder(id = "syn123") |> synDeletePermissions(
        include_self = FALSE,
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("file")
    )

    # Delete permissions for specific entity types (e.g., tables and views)
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("table", "entityview", "materializedview")
    )

    # Dry run example: Log what would be deleted without making changes
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        dry_run = TRUE
    )

    ## End(Not run)

```

synDeletePermissions *Project : delete_permissions*

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can

be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

instance	(Project) The Project instance to operate on.
include_self	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
include_container_content	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if include_self=TRUE.
dry_run	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
show_acl_details	(bool) When dry_run=TRUE, controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.
show_files_in_containers	(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

Project(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself and not files within its subfolder
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
```

```

Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)

```

synDeletePermissions *SubmissionView : delete_permissions*

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
include_self	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
include_container_content	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if include_self=TRUE.
dry_run	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
show_acl_details	(bool) When dry_run=TRUE, controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.
show_files_in_containers	(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

SubmissionView(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself and not files within its subfolder
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)
```



```
# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)
```

synDeletePermissions	Table : delete_permissions
----------------------	----------------------------

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity’s ‘benefactor’. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent’s benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

- instance (Table) The Table instance to operate on.
- include_self (bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
- include_container_content (bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
- recursive (bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.

target_entity_types	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if include_self=TRUE.
dry_run	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
show_acl_details	(bool) When dry_run=TRUE, controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default), shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.
show_files_in_containers	(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

Table(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
```

```

Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself and not files within its subfolder
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE
)

# Delete permissions only for folder entities within this folder recursively
# and their contents
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# Delete permissions only for files within this folder and all subfolders
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("file")
)

# Delete permissions for specific entity types (e.g., tables and views)
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

# Dry run example: Log what would be deleted without making changes
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
  include_container_content = TRUE,
  dry_run = TRUE
)

## End(Not run)

```

Description

Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as `FileEntity` and `Folder` inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

Usage

```
synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, t
```

Arguments

<code>instance</code>	(VirtualTable) The VirtualTable instance to operate on.
<code>include_self</code>	(bool) If TRUE (default), delete the ACL of the current entity. If FALSE, skip deleting the ACL of the current entity.
<code>include_container_content</code>	(bool) If TRUE, delete ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
<code>recursive</code>	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with <code>include_container_content=TRUE</code> to have any effect. Setting <code>recursive=TRUE</code> with <code>include_container_content=FALSE</code> will raise an error. Only works on classes that support the <code>synSyncFromSynapse()</code> function.
<code>target_entity_types</code>	(Optional[list[str]]) Specify which entity types to process when deleting ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file"). This does not affect the entity type of the current entity, which is always processed if <code>include_self=TRUE</code> .
<code>dry_run</code>	(bool) If TRUE, log the changes that would be made instead of actually performing the deletions. When enabled, all ACL deletion operations are simulated and logged at info level. Defaults to FALSE.
<code>show_acl_details</code>	(bool) When <code>dry_run=TRUE</code> , controls whether current ACL details are displayed for entities that will have their permissions changed. If TRUE (default),

shows detailed ACL information. If FALSE, hides ACL details for cleaner output. Has no effect when dry_run=FALSE.

show_files_in_containers

(bool) When dry_run=TRUE, controls whether files within containers are displayed in the preview. If TRUE (default), shows all files. If FALSE, hides files when their only change is benefactor inheritance (but still shows files with local ACLs being deleted). Has no effect when dry_run=FALSE.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues or if the entity already inherits permissions.
- An error may also be raised for any other issues that occur during the process.

Note

The caller must be granted CHANGE_PERMISSIONS access on the Entity to call this method.

Examples

```
## Not run:
## Example 1: Delete permissions for a single entity
library(synapser)
synLogin()

VirtualTable(id = "syn123") |> synDeletePermissions()

## Example 2: Delete permissions recursively for a folder and all its children
library(synapser)
synLogin()

# Delete permissions for this folder only (does not affect children)
Folder(id = "syn123") |> synDeletePermissions()

# Delete permissions for all files and folders directly within this folder,
# but not the folder itself
Folder(id = "syn123") |> synDeletePermissions(
  include_self = FALSE,
  include_container_content = TRUE
)

# Delete permissions for all items in the entire hierarchy (folders and their files)
# Both recursive and include_container_content must be TRUE
Folder(id = "syn123") |> synDeletePermissions(
  recursive = TRUE,
```

```

        include_container_content = TRUE
    )

    # Delete permissions only for folder entities within this folder recursively
    # and their contents
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("folder")
    )

    # Delete permissions only for files within this folder and all subfolders
    Folder(id = "syn123") |> synDeletePermissions(
        include_self = FALSE,
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("file")
    )

    # Delete permissions for specific entity types (e.g., tables and views)
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("table", "entityview", "materializedview")
    )

    # Dry run example: Log what would be deleted without making changes
    Folder(id = "syn123") |> synDeletePermissions(
        recursive = TRUE,
        include_container_content = TRUE,
        dry_run = TRUE
    )

## End(Not run)

```

synDeleteProjectSetting

Folder : delete_project_setting

Description

Delete a project setting by its setting ID.

Usage

```
synDeleteProjectSetting(instance, setting_id, synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.
 setting_id (character) The ID of the project setting to delete.
 synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Using this function
# Delete the upload settings for a folder:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
setting = folder |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  folder |> synDeleteProjectSetting(setting_id = setting$id)
}

## End(Not run)
```

synDeleteProjectSetting

Project : delete_project_setting

Description

Delete a project setting by its setting ID.

Usage

```
synDeleteProjectSetting(instance, setting_id, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.
 setting_id (character) The ID of the project setting to delete.
 synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Using this function
# Delete the upload settings for a project:
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
setting = project |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  project |> synDeleteProjectSetting(setting_id = setting$id)
}

## End(Not run)
```

synDeleteRows	<i>Table : delete_rows</i>
---------------	----------------------------

Description

Delete rows from a table given a query or a data frame to select rows. The query at a minimum must select the ROW_ID and ROW_VERSION columns. If you want to inspect the data that will be deleted ahead of time you may use the [synQuery\(\)](#) function to get the data. The data frame must at least contain the ROW_ID and ROW_VERSION columns. And ROW_ETAG column is also required if the entity is one of the following: EntityView, Dataset, DatasetCollection, or SubmissionView. If both query and df are provided, the query will be used.

Usage

```
synDeleteRows(instance, query=NULL, df=NULL, job_timeout=600L, synapse_client=NULL)
```

Arguments

- instance (Table) The Table instance to operate on.
- query (Optional[str]) The query to select the rows to delete. The query at a minimum must select the ROW_ID and ROW_VERSION columns. See this document that describes the expected syntax of the query: <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/we>

df	(Optional[data.frame]) A data frame that contains the rows to delete. The data frame must at least contain the ROW_ID and ROW_VERSION columns. If the entity is one of the following: EntityView, Dataset, DatasetCollection, or SubmissionView then the data frame must also contain the ROW_ETAG column. See this document that describes the expected columns of the data frame: < https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/table/Row.html >
job_timeout	(int) The amount of time to wait for table updates to complete before a timeout error is thrown. The default is 600 seconds.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of your query or data frame for the rows that were deleted from the table.

Examples

```
## Not run:
## Example 1: Selecting a row to delete
# This example shows how you may select a row to delete from a table.
library(synapser)
synLogin()

# Also works with Dataset
Table(id = "syn1234") |> synDeleteRows(query = "SELECT ROW_ID, ROW_VERSION FROM syn1234 WHERE foo = 'asdf'")

## Example 2: Selecting all rows that contain a null value
# This example shows how you may select a row to delete from a table where
# a column has a null value.
library(synapser)
synLogin()

# Also works with Dataset
Table(id = "syn1234") |> synDeleteRows(query = "SELECT ROW_ID, ROW_VERSION FROM syn1234 WHERE foo is null")

## Example 3: Selecting rows to delete using a data frame
# This example shows how you may select a row to delete from a table based on a data frame.
library(synapser)
synLogin()

# Creating a data frame that contains the rows to delete.
# In this example, we create a data frame that specifies the first two rows of the table for deletion.
# Assuming no changes have been made to the table so the ROW_VERSION is 1L.

df = data.frame(ROW_ID = c(1L, 2L), ROW_VERSION = c(1L, 1L ))
Table(id = "syn1234") |> synDeleteRows(df = df)

## End(Not run)
```

 synDisassociateActivityFromEntity

Activity : disassociate_from_entity

Description

Disassociate the Activity from the parent entity. This is the first step in deleting the Activity. If you have other entities that are associated with this Activity you must disassociate them by calling this method on them as well.

Usage

```
synDisassociateActivityFromEntity(cls, parent, synapse_client=NULL)
```

Arguments

cls	(Activity) The Activity class (or an existing Activity instance) to dispatch this call to.
parent	(Union[Table, File, str]) The parent entity this activity is associated with.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

An error is raised if the parent does not have an ID.

Examples

```
## Not run:
## Using this function
# Remove the Activity from the entity it is associated with. The Activity
# itself is deleted once it is no longer associated with any entity.
library(synapser)
synLogin()

Activity() |> synDisassociateActivityFromEntity(parent = "syn123")

## End(Not run)
```

synDownloadListAdd	<i>synDownloadListAdd</i>
--------------------	---------------------------

Description

Add files to the Synapse download list.

If a file is added with no version specified, the latest version will be downloaded.

Usage

```
synDownloadListAdd(files, synapse_client=NULL)
```

Arguments

files	(list[DownloadListItem]) List of DownloadListItem objects identifying the file versions to add.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The number of files added.

Examples

```
## Not run:
## Add files to the download list
# Add specific file versions to the cart.
library(synapser)
synLogin()

count = synDownloadListAdd(list(
  DownloadListItem(file_entity_id = "syn123", version_number = 1L),
  DownloadListItem(file_entity_id = "syn456", version_number = 2L)
))

## End(Not run)
```

synDownloadListClear *synDownloadListClear*

Description

Clear all files from the Synapse download list (cart).

Usage

```
synDownloadListClear(synapse_client=NULL)
```

Arguments

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Clear the download list
# Remove all files from the cart.
library(synapser)
synLogin()

synDownloadListClear()

## End(Not run)
```

synDownloadListFiles *synDownloadListFiles*

Description

Download all files in the Synapse download list (cart) to a local directory.

Files are downloaded individually. The cart is not packaged into a zip because download lists can exceed 100 GB. Only successfully downloaded files are removed from the cart after the full pass completes, so interrupted runs are safely resumable.

Files that cannot be accessed or fail to download are left in the cart and recorded with an error value in the result manifest.

Usage

```
synDownloadListFiles(download_location=NULL, parallel=FALSE, max_concurrent=10L, synapse_client=NULL)
```

Arguments

download_location	(Optional[character]) Directory to download files to. Defaults to the current working directory.
parallel	(logical) If TRUE, files are downloaded concurrently, up to max_concurrent at a time. If FALSE (default), files are downloaded sequentially.
max_concurrent	(integer) Maximum number of files to download concurrently when parallel=TRUE. Defaults to 10. Has no effect when parallel=FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

Path to the result manifest CSV, which contains all original manifest columns plus path (local file path) and error (error message or empty string) columns.

Errors

- An error is raised if the manifest async job fails or the cart is empty ("No files available for download").
- An error is also raised if the manifest job completes but produces no local file, or if the downloaded CSV has no headers or contains reserved column names ("path" or "error").

Examples

```
## Not run:
## Download all files in the cart
# Download all files in the user's download list to a local directory.
library(synapser)
synLogin()

manifest_path = synDownloadListFiles(download_location = "../data")

## End(Not run)
```

synDownloadListManifest

synDownloadListManifest

Description

Generate and download the manifest CSV for the current cart contents.

Submits an async job to Synapse to generate the manifest, then downloads the resulting CSV. The manifest contains the same columns as the zip manifest downloaded from the Synapse web UI.

Usage

```
synDownloadListManifest(csv_table_descriptor=NULL, destination=".", synapse_client=NULL)
```

Arguments

`csv_table_descriptor` (Optional[CsvTableDescriptor]) Optional CsvTableDescriptor controlling the format of the generated CSV (separator, quote character, escape character, line ending, and whether the first line is a header). When omitted the Synapse defaults are used.

`destination` (character) Directory to download the manifest CSV to. Defaults to the current working directory.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

Path to the downloaded manifest CSV.

Errors

Raises an error if the async job completes without producing a manifest.

Examples

```
## Not run:
## Get the download list manifest
# Inspect the cart contents before downloading.
library(synapser)
synLogin()

manifest_path = synDownloadListManifest()

## End(Not run)
```

```
synDownloadListRemove synDownloadListRemove
```

Description

Remove files from the Synapse download list.

If a file was added with a version specified, then that version must be specified to remove it. If a file was added with no version specified, then no version must be specified to remove it.

Usage

```
synDownloadListRemove(files, synapse_client=NULL)
```

Arguments

- files (list[DownloadListItem]) List of [DownloadListItem](#) objects identifying the file versions to remove.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The number of files removed.

Examples

```
## Not run:
## Remove files from the download list
# Remove specific file versions from the cart.
library(synapser)
synLogin()

count = synDownloadListRemove(list(
  DownloadListItem(file_entity_id = "syn123", version_number = 1L)
))

## End(Not run)
```

synFindEntityId	<i>synFindEntityId</i>
-----------------	------------------------

Description

Find an Entity given its name and parent.

Usage

```
synFindEntityId(name, parent=NULL, synapse_client=NULL)
```

Arguments

- name (str) Name of the entity to find.
- parent (Optional[Union[str, Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable]]) An Entity object (Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable) or the Id of an entity as a string. Omit if searching for a Project by name.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The Entity ID or NULL if not found.

Examples

```
## Not run:
## Example 1: Finding a project by name
# Find a project using only its name:
library(synapser)
synLogin()

project_id = synFindEntityId(name = "My Project")
if (!is.null(project_id)) {
  print(sprintf("Found project: %s", project_id))
} else {
  print("Project not found")
}

## Example 2: Finding an entity within a parent
# Find a file within a specific folder:
library(synapser)
synLogin()

file_id = synFindEntityId(
  name = "my_data.csv",
  parent = "syn123456" # Parent folder ID
)
if (!is.null(file_id)) {
  print(sprintf("Found file: %s", file_id))
} else {
  print("File not found in folder")
}

## Example 3: Using with entity objects
# Find an entity using an entity object as the parent:
library(synapser)
synLogin()

parent_folder = Folder(id = "syn123456")
entity_id = synFindEntityId(
  name = "analysis_results.txt",
  parent = parent_folder
)
print(sprintf("Entity ID: %s", entity_id))

## End(Not run)
```


Description

Recursively loop over all of the already retrieved files and folders and return a list of all files in the container.

Usage

```
synFlattenFileList(instance)
```

Arguments

instance (Folder) The Folder instance to operate on.

Value

A list of all files in the container.

Examples

```
## Not run:
## Using this function
# synFlattenFileList only looks at files and folders that have already been
# retrieved, so sync the folder from Synapse first.
library(synapser)
synLogin()

my_folder = Folder(id = "syn123") |> synSyncFromSynapse(download_file = FALSE)

all_files = my_folder |> synFlattenFileList()
for (file in all_files) {
  print(sprintf("%s (%s)", file$name, file$id))
}

## End(Not run)
```

```
synFlattenFileList      Project : flatten_file_list
```

Description

Recursively loop over all of the already retrieved files and folders and return a list of all files in the container.

Usage

```
synFlattenFileList(instance)
```

Arguments

instance (Project) The Project instance to operate on.

Value

A list of all files in the container.

Examples

```
## Not run:
## Using this function
# synFlattenFileList only looks at files and folders that have already been
# retrieved, so sync the project from Synapse first.
library(synapser)
synLogin()

my_project = Project(id = "syn123") |> synSyncFromSynapse(download_file = FALSE)

all_files = my_project |> synFlattenFileList()
for (file in all_files) {
  print(sprintf("%s (%s)", file$name, file$id))
}

## End(Not run)
```

synGenerateSyncManifest

Folder : generate_sync_manifest

Description

Walk a local directory, mirror its folder hierarchy under this container in Synapse, and write a CSV manifest ready for [synSyncToSynapse\(\)](#).

The manifest has two columns: path (absolute, symlink-resolved) and parentId (the Synapse ID of the file's containing folder). Existing Synapse folders with matching names and parents are reused. Directory symlinks inside directory_path are not followed; file symlinks record the symlink path and upload the target's contents. Zero-byte files are skipped with a warning — Synapse rejects empty files. I/O errors during walk are logged and skipped; an empty source directory produces a warning and a header-only manifest.

Usage

```
synGenerateSyncManifest(instance, directory_path, manifest_path, synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.

directory_path (character) Path to the local directory to be pushed to Synapse.

manifest_path (character) Path where the generated manifest CSV will be written.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if this container's id is NULL, or if directory_path does not exist or is not a directory, or if this container's id exists in Synapse but is not a Folder or Project.
- An error is also raised if this container's id does not exist in Synapse.

Examples

```
## Not run:
## Generate a manifest and upload the files
# Mirror ./my_data under a Synapse folder and then upload it.
library(synpser)
synLogin()

folder = Folder(id = "syn12345")
folder |> synGenerateSyncManifest(
  directory_path = "./my_data",
  manifest_path = "manifest.csv"
)
folder |> synSyncToSynapse(manifest_path = "manifest.csv")

## End(Not run)
```

`synGenerateSyncManifest`*Project : generate_sync_manifest*

Description

Walk a local directory, mirror its folder hierarchy under this container in Synapse, and write a CSV manifest ready for [synSyncToSynapse\(\)](#).

The manifest has two columns: path (absolute, symlink-resolved) and parentId (the Synapse ID of the file's containing folder). Existing Synapse folders with matching names and parents are reused. Directory symlinks inside directory_path are not followed; file symlinks record the symlink path and upload the target's contents. Zero-byte files are skipped with a warning — Synapse rejects empty files. I/O errors during walk are logged and skipped; an empty source directory produces a warning and a header-only manifest.

Usage

```
synGenerateSyncManifest(instance, directory_path, manifest_path, synapse_client=NULL)
```

Arguments

- instance (Project) The Project instance to operate on.
- directory_path (character) Path to the local directory to be pushed to Synapse.
- manifest_path (character) Path where the generated manifest CSV will be written.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

- An error is raised if this container's id is NULL, or if directory_path does not exist or is not a directory, or if this container's id exists in Synapse but is not a Folder or Project.
- An error is also raised if this container's id does not exist in Synapse.

Examples

```
## Not run:
## Generate a manifest and upload the files
# Mirror ./my_data under a Synapse project and then upload it.
library(synapser)
synLogin()

project = Project(id = "syn12345")
project |> synGenerateSyncManifest(
  directory_path = "./my_data",
  manifest_path = "manifest.csv"
)
project |> synSyncToSynapse(manifest_path = "manifest.csv")

## End(Not run)
```

synGet	<i>synGet</i>
--------	---------------

Description

Factory method to retrieve any Synapse entity by its ID or by name and parent ID.

This method serves as a unified interface for retrieving any type of Synapse entity without needing to know the specific entity type beforehand. It automatically determines the entity type and returns the appropriate model instance.

You can retrieve entities in two ways:

1. By providing a synapse_id directly
2. By providing entity_name and optionally parent_id for lookup

Usage

```
synGet(synapse_id=NULL, entity_name=NULL, parent_id=NULL, version_number=NULL, activity_options=NULL,
```

Arguments

<code>synapse_id</code>	(Optional[str]) The Synapse ID of the entity to retrieve (e.g., 'syn123456'). Mutually exclusive with <code>entity_name</code> .
<code>entity_name</code>	(Optional[str]) The name of the entity to find. Must be used with this approach instead of <code>synapse_id</code> . When looking up projects, <code>parent_id</code> should be NULL.
<code>parent_id</code>	(Optional[str]) The parent entity ID when looking up by name. Set to NULL when looking up projects by name. Only used with <code>entity_name</code> .
<code>version_number</code>	(Optional[int]) The specific version number of the entity to retrieve. Only applies to versionable entities (File, Table, Dataset). If not specified, the most recent version will be retrieved. Ignored for other entity types.
<code>activity_options</code>	(Optional[ActivityOptions]) ActivityOptions to apply. Can be applied to any entity type to include activity information.
<code>file_options</code>	(Optional[FileOptions]) FileOptions to apply. Only applies to File entities. Ignored for other entity types.
<code>table_options</code>	(Optional[TableOptions]) TableOptions to apply. Only applies to Table-like entities (Table, Dataset, EntityView, MaterializedView, SubmissionView, VirtualTable, DatasetCollection). Ignored for other entity types.
<code>link_options</code>	(Optional[LinkOptions]) LinkOptions to apply. Only applies when the entity is a Link. Ignored for other entity types.
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The appropriate Synapse entity model instance based on the entity type.

Errors

- An error is raised if both `synapse_id` and `entity_name` are provided, or if neither is provided.
- An error is raised if `entity_name` is provided without this being a valid lookup scenario.
- An error is raised if the `synapse_id` is not a valid Synapse ID format.

Note

When using `entity_name` lookup:

- For projects: leave `parent_id` = NULL.
- For all other entities: provide the `parent_id` of the containing folder/project.

Examples

```
## Not run:
## Example 1: Retrieving entities by ID
# Get any entity by Synapse ID:
library(synapser)
synLogin()

# Works for any entity type
entity = synGet(synapse_id = "syn123456")

# The returned object will be the appropriate type
if (inherits(entity, "File")) {
  print(sprintf("File: %s", entity$name))
} else if (inherits(entity, "Project")) {
  print(sprintf("Project: %s", entity$name))
}

## Example 2: Retrieving entities by name
# Get an entity by name and parent:
library(synapser)
synLogin()

# Get a file by name within a folder
entity = synGet(
  entity_name = "my_file.txt",
  parent_id = "syn123456"
)

# Get a project by name (parent_id = NULL)
project = synGet(
  entity_name = "My Project",
  parent_id = NULL
)

## Example 3: Retrieving a specific version
# Get a specific version of a versionable entity:
library(synapser)
synLogin()

entity = synGet(synapse_id = "syn123456", version_number = 2L)

## Example 4: Retrieving a file with custom options
# Get file metadata with specific download options:
library(synapser)
synLogin()

file_entity = synGet(
  synapse_id = "syn123456",
  activity_options = ActivityOptions(include_activity = TRUE),
  file_options = FileOptions(
    download_file = FALSE
  )
)
```

```
)

## Example 5: Retrieving a table with activity and columns
# Get table with activity and column information:
library(synapser)
synLogin()

table_entity = synGet(
  synapse_id = "syn123456",
  activity_options = ActivityOptions(include_activity = TRUE),
  table_options = TableOptions(include_columns = TRUE)
)

## Example 6: Following links
# Get the target of a link entity:
library(synapser)
synLogin()

target_entity = synGet(
  synapse_id = "syn123456",
  link_options = LinkOptions(follow_link = TRUE)
)

## Example 7: Working with Link entities
# Get a Link entity without following it:
library(synapser)
synLogin()

# Get the link entity itself
link_entity = synGet(
  synapse_id = "syn123456", # Example link ID
  link_options = LinkOptions(follow_link = FALSE)
)
print(sprintf("Link: %s -> %s", link_entity$name, link_entity$target_id))

# Then follow the link to get the target
target_entity = synGet(
  synapse_id = "syn123456",
  link_options = LinkOptions(follow_link = TRUE)
)
print(sprintf("Target: %s (type: %s)", target_entity$name, class(target_entity)[1]))

## Example 8: Comprehensive File options
# Download file with custom location and collision handling:
library(synapser)
synLogin()

file_entity = synGet(
  synapse_id = "syn123456",
  file_options = FileOptions(
    download_file = TRUE,
    download_location = "/path/to/download/",
    if_collision = "overwrite.local"
```

```

    )
  )
  print(sprintf("Downloaded file: %s to %s", file_entity$name, file_entity$path))

## Example 9: Table options for table-like entities
# Get table entities with column information:
library(synapser)
synLogin()

# Works for Table, Dataset, EntityView, MaterializedView,
# SubmissionView, VirtualTable, and DatasetCollection
table_entity = synGet(
  synapse_id = "syn123456", # Example table ID
  table_options = TableOptions(include_columns = TRUE)
)
print(sprintf("Table: %s with %d columns", table_entity$name, length(table_entity$columns)))

## Example 10: Combining multiple options
# Get a File with both activity and custom download options:
library(synapser)
synLogin()

file_entity = synGet(
  synapse_id = "syn123456",
  activity_options = ActivityOptions(include_activity = TRUE),
  file_options = FileOptions(
    download_file = FALSE
  )
)
print(sprintf("File: %s (activity included: %s)", file_entity$name, !is.null(file_entity$activity)))

## Example 11: Working with entity instances
# Pass an existing entity instance to refresh or apply new options:
library(synapser)
synLogin()

# Get an entity first
entity = synGet(synapse_id = "syn123456")
print(sprintf("Original entity: %s", entity$name))

# Then use the entity instance to get it again with different options
refreshed_entity = entity |> synGet(file_options = FileOptions(download_file = FALSE))
print(sprintf("Refreshed entity: %s (download_file: %s)", refreshed_entity$name, refreshed_entity$download_file))

## End(Not run)

```

synGetAcl

Dataset : get_acl

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance (Dataset) The Dataset instance to operate on.

principal_id (integer) Identifier of a user or group (defaults to PUBLIC users)

check_benefactor (logical) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty list.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a Dataset
library(synapser)
synLogin()

acl = Dataset(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the Dataset
acl = Dataset(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the Dataset, not the one it inherits
acl = Dataset(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl	<i>DatasetCollection : get_acl</i>
-----------	------------------------------------

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
principal_id	(integer) Identifier of a user or group (defaults to PUBLIC users)
check_benefactor	(logical) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty list.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a DatasetCollection
library(synapser)
synLogin()

acl = DatasetCollection(id = "syn123") |> synGetAcl()
print(acl)
```

```
# Get the permissions a specific user or team has on the DatasetCollection
acl = DatasetCollection(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the DatasetCollection, not the one it inherits
acl = DatasetCollection(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

EntityView : get_acl

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance (EntityView) The EntityView instance to operate on.

principal_id (int) Identifier of a user or group (defaults to PUBLIC users)

check_benefactor (bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a EntityView
library(synapser)
synLogin()

acl = EntityView(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the EntityView
acl = EntityView(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the EntityView, not the one it inherits
acl = EntityView(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

*Evaluation : get_acl***Description**

Get the access control list (ACL) governing this evaluation.

Usage

```
synGetAcl(instance, synapse_client=NULL)
```

Arguments

`instance` (Evaluation) The Evaluation instance to operate on.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The **AccessControlList** response object as a named list.

Errors

- An error is raised if the evaluation's id is not set.
- An error is also raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Get the ACL for an evaluation
library(synapser)
synLogin()

PROJECT_ID = "syn123456" # replace with your project ID

# Store a new evaluation in the project first
evaluation = Evaluation(
  name = "My Challenge Evaluation",
  description = "Evaluation for my data challenge",
  content_source = PROJECT_ID,
  submission_instructions_message = "Submit CSV files only",
  submission_receipt_message = "Thank you for your submission!"
) |> synStore()

# Get the evaluation back from Synapse
evaluations = synGetEvaluationsByProject(
  project_id = PROJECT_ID,
  evaluation_ids = list(evaluation$id)
)
evaluation = evaluations[[1]]

# Get the ACL for the evaluation
acl = evaluation |> synGetAcl()
acl

## End(Not run)
```

synGetAcl

File : *get_acl*

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance	(File) The File instance to operate on.
principal_id	(int) Identifier of a user or group (defaults to PUBLIC users)
check_benefactor	(bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a File
library(synpser)
synLogin()

acl = File(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the File
acl = File(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the File, not the one it inherits
acl = File(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

Folder : get_acl

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance	(Folder) The Folder instance to operate on.
principal_id	(int) Identifier of a user or group (defaults to PUBLIC users)
check_benefactor	(bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a Folder
library(synapseR)
synLogin()

acl = Folder(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the Folder
acl = Folder(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the Folder, not the one it inherits
acl = Folder(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl	<i>MaterializedView : get_acl</i>
-----------	-----------------------------------

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance	(MaterializedView) The MaterializedView instance to operate on.
principal_id	(int) Identifier of a user or group (defaults to PUBLIC users)
check_benefactor	(bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a MaterializedView
library(synpser)
synLogin()

acl = MaterializedView(id = "syn123") |> synGetAcl()
print(acl)
```



```
# Get the permissions a specific user or team has on the MaterializedView
acl = MaterializedView(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the MaterializedView, not the one it inherits
acl = MaterializedView(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

Organization : get_acl

Description

Gets the ACL for this organization

Usage

```
synGetAcl(instance, synapse_client=NULL)
```

Arguments

instance (Organization) The Organization instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A named list in the form of this response: [AccessControlList](#)

Examples

```
## Not run:
## Get the ACL for a Organization
library(synapser)
synLogin()

org = Organization(name = "dpetest")
acl = org |> synGetAcl()

etag = acl[["etag"]]
print(etag)
resource_access = acl[["resourceAccess"]]
for (item in resource_access) {
  principal_id = item[["principalId"]]
  print(principal_id)
  access_types = item[["accessType"]]
}
```

```

        print(access_types)
    }

    ## End(Not run)

```

synGetAcl

Project : get_acl

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

principal_id (int) Identifier of a user or group (defaults to PUBLIC users)

check_benefactor (bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a Project
library(synapser)
synLogin()

acl = Project(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the Project
acl = Project(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the Project, not the one it inherits
acl = Project(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

*SubmissionView : get_acl***Description**

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
principal_id	(int) Identifier of a user or group (defaults to PUBLIC users)
check_benefactor	(bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity’s benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a SubmissionView
library(synapser)
synLogin()

acl = SubmissionView(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the SubmissionView
acl = SubmissionView(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the SubmissionView, not the one it inherits
acl = SubmissionView(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl	<i>Table : get_acl</i>
-----------	------------------------

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

- instance (Table) The Table instance to operate on.
- principal_id (int) Identifier of a user or group (defaults to PUBLIC users)
- check_benefactor (bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the permissions the PUBLIC group has on a Table
library(synpser)
synLogin()

acl = Table(id = "syn123") |> synGetAcl()
print(acl)

# Get the permissions a specific user or team has on the Table
acl = Table(id = "syn123") |> synGetAcl(principal_id = 273948L)
print(acl)

# Only look at the ACL set directly on the Table, not the one it inherits
acl = Table(id = "syn123") |> synGetAcl(principal_id = 273948L, check_benefactor = FALSE)
print(acl)

## End(Not run)
```

synGetAcl

VirtualTable : get_acl

Description

Get the **ACL** that a user or group has on an Entity.

Usage

```
synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)
```

Arguments

`instance` (VirtualTable) The VirtualTable instance to operate on.

`principal_id` (int) Identifier of a user or group (defaults to PUBLIC users)

`check_benefactor` (bool) If TRUE (default), check the benefactor for the entity to get the ACL. If FALSE, only check the entity itself. This is useful for checking the ACL of an entity that has local sharing settings, but you want to check the ACL of the entity itself and not the benefactor it may inherit from.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An array containing some combination of READ, UPDATE, CREATE, DELETE, DOWNLOAD, MODERATE, CHANGE_PERMISSIONS, CHANGE_SETTINGS, or an empty array.

Note

If the entity does not have local sharing settings, or ACL set directly on it, this will look up the ACL on the benefactor of the entity. The benefactor is the entity that the current entity inherits its permissions from. The benefactor is usually the parent entity, but it can be any ancestor in the hierarchy. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project or Folder. If the entity already has local sharing settings, the benefactor would be itself.

Examples

```
## Not run:
## Using this function
# Get the ACL for a specific principal on a VirtualTable
acl = VirtualTable(id = "syn123") |> synGetAcl(principal_id = 273948L)

## End(Not run)
```

`synGetAllEvaluations` *Evaluation : get_all_evaluations*

Description

Get a list of all Evaluations, within a given range.

Usage

```
synGetAllEvaluations(access_type=NULL, active_only=NULL, evaluation_ids=NULL, offset=NULL, limit=NULL)
```

Arguments

access_type	(Optional[str]) The type of access for the user to filter for, optional and defaults to "READ". See ACCESS_TYPE for the available values.
active_only	(Optional[bool]) If TRUE then return only those evaluations with rounds defined and for which the current time is in one of the rounds.
evaluation_ids	(Optional[list[str]]) An optional list of evaluation IDs to which the response is limited.
offset	(Optional[int]) The offset index determines where this page will start from. An index of 0 is the first entity. When NULL it will default to 0.
limit	(Optional[int]) Limits the number of entities that will be fetched for this page. When NULL it will default to 10.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of all evaluations, as Evaluation objects.

Errors

An error is raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Example 1: Get all evaluations the user has at least READ access to
# A default call will return evaluations where the user has read access, without needing to specify access type.
library(synapser)
synLogin()

all_evaluations = synGetAllEvaluations()
all_evaluations

## Example 2: Get only active evaluations with a limit
library(synapser)
synLogin()

active_evaluations = synGetAllEvaluations(
  active_only = TRUE,
  limit = 20L
)
active_evaluations

## Example 3: Get specific evaluations by ID
library(synapser)
synLogin()

specific_evaluations = synGetAllEvaluations(
  evaluation_ids = list("9999991", "9999992")
)
```

```

    )
    specific_evaluations

## End(Not run)

```

synGetAllSubmissionStatuses

SubmissionStatus : get_all_submission_statuses

Description

Gets a collection of SubmissionStatuses to a specified Evaluation.

Usage

```
synGetAllSubmissionStatuses(evaluation_id, status=NULL, limit=10L, offset=0L, synapse_client=NULL)
```

Arguments

evaluation_id (character) The ID of the specified Evaluation.

status (Optional[character]) Optionally filter submission statuses by status.

limit (integer) Limits the number of entities that will be fetched for this page. When NULL it will default to 10, max value 100. Defaults to 10.

offset (integer) The offset index determines where this page will start from. An index of 0 is the first entity. Defaults to 0.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of SubmissionStatus objects for the evaluation queue.

Examples

```

## Not run:
## Getting all submission statuses for an evaluation
# Retrieve a list of submission statuses for a specific evaluation, optionally filtered by status.
library(synapser)
synLogin()

statuses = synGetAllSubmissionStatuses(
  evaluation_id = "9999999",
  status = "SCORED",
  limit = 50L
)
print(sprintf("Found %d submission statuses", length(statuses)))
for (status in statuses) {
  print(sprintf("Status ID: %s, Status: %s", status$id, status$status))
}

```



```

    }

    ## End(Not run)

```

synGetAttachment	<i>WikiPage : get_attachment</i>
------------------	----------------------------------

Description

Download the wiki page attachment to a local file or return the URL.

Usage

```
synGetAttachment(instance, file_name, download_file=TRUE, download_location=NULL, synapse_client=NULL)
```

Arguments

instance (WikiPage) The WikiPage instance to operate on.

file_name (str) The name of the file to get. The file name can be in either non-gzipped or gzipped format.

download_file (bool) Whether associated files should be downloaded. Default is TRUE.

download_location (Optional[str]) The directory to download the file to. Required if **download_file** is TRUE.

synapse_client (Optional[Synapse]) Optionally provide a Synapse client.

Value

If **download_file** is TRUE, the attachment file will be downloaded to the **download_location**. Otherwise, the URL will be returned.

Errors

An error is raised if **owner_id** or **id** is not provided.

Examples

```

## Not run:
## Example 1: Get the attachment URL for a wiki page
# This example shows how to get the attachment file or URL for a wiki page.
attachment_file_or_url = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetAttachment(file_name = "attachment_file_or_url")
cat("Attachment URL:", attachment_file_or_url, "\n")

## Example 2: Download the attachment file for a wiki page
# This example shows how to download the attachment file for a wiki page.
attachment_file_path = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetAttachment(file_name = "attachment_file_path")
cat("Attachment file path:", attachment_file_path, "\n")

## End(Not run)

```

```
synGetAttachmentHandles
```

```
WikiPage : get_attachment_handles
```

Description

Get the file handles of all attachments on this wiki page.

Usage

```
synGetAttachmentHandles(instance, synapse_client=NULL)
```

Arguments

`instance` (WikiPage) The WikiPage instance to operate on.
`synapse_client` (Optional[Synapse]) Optionally provide a Synapse client.

Value

The list of FileHandles for all file attachments of this WikiPage.

Errors

An error is raised if owner_id or id is not provided.

Examples

```
## Not run:
## Get the file handles of all attachments on a wiki page
# This example shows how to get the file handles of all attachments on a wiki page.
attachment_handles = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetAttachmentHandles()
cat("Attachment handles:\n")
attachment_handles$list

## End(Not run)
```

```
synGetAttachmentPreview
```

```
WikiPage : get_attachment_preview
```

Description

Download the wiki page attachment preview to a local file or return the URL.

Usage

```
synGetAttachmentPreview(instance, file_name, download_file=TRUE, download_location=NULL, synapse_client=NULL)
```

Arguments

instance (WikiPage) The WikiPage instance to operate on.

file_name (str) The name of the file to get. The file name can be in either non-gzipped or gzipped format.

download_file (bool) Whether associated files should be downloaded. Default is TRUE.

download_location (Optional[str]) The directory to download the file to. Required if download_file is TRUE.

synapse_client (Optional[Synapse]) Optionally provide a Synapse client.

Value

If download_file is TRUE, the attachment preview file will be downloaded to the download_location. Otherwise, the URL will be returned.

Errors

An error is raised if owner_id or id is not provided.

Examples

```
## Not run:
## Example 1: Get the attachment preview URL for a wiki page
# This example shows how to get the attachment preview URL for a wiki page.
# Instead of using the file_name from the attachmenhandle response when isPreview=TRUE, you should use the original file_name
# The downloaded file will still be named according to the file_name provided in the response when isPreview=TRUE
attachment_preview_url = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetAttachmentPreview(file_name = file_name)
cat("Attachment preview URL:", attachment_preview_url, "\n")

## Example 2: Download the attachment preview file for a wiki page
# This example shows how to download the attachment preview file for a wiki page.
attachment_preview_file_path = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetAttachmentPreview(file_name = file_name)
cat("Attachment preview file path:", attachment_preview_file_path, "\n")

## End(Not run)
```

synGetAvailableEvaluations

Evaluation : get_available_evaluations

Description

Get a list of Evaluations to which the user has SUBMIT permission, within a given range.

Usage

```
synGetAvailableEvaluations(active_only=NULL, evaluation_ids=NULL, offset=NULL, limit=NULL, synapse_client=NULL)
```

Arguments

active_only	(Optional[bool]) If TRUE then return only those evaluations with rounds defined and for which the current time is in one of the rounds.
evaluation_ids	(Optional[list[str]]) An optional list of evaluation IDs to which the response is limited.
offset	(Optional[int]) The offset index determines where this page will start from. An index of 0 is the first entity. When NULL it will default to 0.
limit	(Optional[int]) Limits the number of entities that will be fetched for this page. When NULL it will default to 10.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of available evaluations, as Evaluation objects.

Errors

An error is raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Example 1: Get all evaluations where the current user has SUBMIT permission
# A default call will return evaluations where the user has SUBMIT permission, without needing to specify access t
library(synapser)
synLogin()

available_evaluations = synGetAvailableEvaluations()
available_evaluations

## Example 2: Get only active evaluations where the current user has SUBMIT permission
library(synapser)
synLogin()

active_available_evaluations = synGetAvailableEvaluations(
  active_only = TRUE
)
active_available_evaluations

## Example 3: Get the first 5 evaluations where the current user has SUBMIT permission
library(synapser)
synLogin()

limited_evaluations = synGetAvailableEvaluations(
  limit = 5L
)
limited_evaluations

## End(Not run)
```

synGetChatHistory	<i>Agent : get_chat_history</i>
-------------------	---------------------------------

Description

Gets the chat history for the current session.

Usage

```
synGetChatHistory(instance)
```

Arguments

instance	(Agent) The Agent instance to operate on.
----------	---

Value

NULL

Examples

```
## Not run:
## Get the chat history for the current session.
# First, send a prompt to the agent.
# Then, retrieve the chat history for the current session by calling synGetChatHistory().
library(synapser)
synLogin()

my_agent = Agent()
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)
print(my_agent |> synGetChatHistory())

## End(Not run)
```

synGetEvaluationsByProject	<i>Evaluation : get_evaluations_by_project</i>
----------------------------	--

Description

Get Evaluations tied to a project.

Usage

```
synGetEvaluationsByProject(project_id, access_type=NULL, active_only=NULL, evaluation_ids=NULL, offset=0, limit=10, synapse_client=NULL)
```

Arguments

<code>project_id</code>	(str) The ID of the project (e.g., "syn123456").
<code>access_type</code>	(Optional[str]) The type of access for the user to filter for, optional and defaults to "READ". See ACCESS_TYPE for the available values.
<code>active_only</code>	(Optional[bool]) If TRUE then return only those evaluations with rounds defined and for which the current time is in one of the rounds.
<code>evaluation_ids</code>	(Optional[list[str]]) An optional list of evaluation IDs to which the response is limited.
<code>offset</code>	(Optional[int]) The offset index determines where this page will start from. An index of 0 is the first entity. When NULL it will default to 0.
<code>limit</code>	(Optional[int]) Limits the number of entities that will be fetched for this page. When NULL it will default to 10.
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of Evaluations tied to the project, as Evaluation objects.

Errors

An error is raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Example 1: Get all evaluations for a project
# The user must have at least READ access to the evaluations to retrieve the evaluations from a given project.
library(synapser)
synLogin()

project_evaluations = synGetEvaluationsByProject(
  project_id = "syn123456"
)
project_evaluations

## Example 2: Get only active evaluations for a project
library(synapser)
synLogin()

active_project_evaluations = synGetEvaluationsByProject(
  project_id = "syn123456",
  active_only = TRUE
)
active_project_evaluations
```

```
## Example 3: Get a limited set of evaluations for a project
library(synapser)
synLogin()

limited_project_evaluations = synGetEvaluationsByProject(
  project_id = "syn123456",
  limit = 5L
)
limited_project_evaluations

## End(Not run)
```

synGetEvaluationSubmissionBundles

SubmissionBundle : get_evaluation_submission_bundles

Description

Retrieves bundled Submissions and SubmissionStatuses for a given Evaluation.

Usage

```
synGetEvaluationSubmissionBundles(cls, evaluation_id, status=NULL, synapse_client=NULL)
```

Arguments

<code>cls</code>	(SubmissionBundle) The SubmissionBundle class (or an existing Submission-Bundle instance) to dispatch this call to.
<code>evaluation_id</code>	(character) The ID of the specified Evaluation.
<code>status</code>	(Optional[character]) Optionally filter submission bundles by status.
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields [SubmissionBundle](#) objects as they are retrieved from the API. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one [SubmissionBundle](#) at a time.

Note

The caller must be granted the `READ_PRIVATE_SUBMISSION` [access type](#) on the specified Evaluation.

Examples

```
## Not run:
## Getting submission bundles for an evaluation
library(synapser)
synLogin()

bundles = as.list(SubmissionBundle() |> synGetEvaluationSubmissionBundles(
  evaluation_id = "9999999",
  status = "SCORED"
))
print(sprintf("Found %d submission bundles", length(bundles)))
for (bundle in bundles) {
  submission_id = if (is.null(bundle$submission)) "N/A" else bundle$submission$id
  print(sprintf("Submission ID: %s", submission_id))
}

## End(Not run)
```

synGetEvaluationSubmissions

Submission : get_evaluation_submissions

Description

Retrieves all Submissions for a specified Evaluation queue.

Usage

```
synGetEvaluationSubmissions(cls, evaluation_id, status=NULL, synapse_client=NULL)
```

Arguments

cls	(Submission) The Submission class (or an existing Submission instance) to dispatch this call to.
evaluation_id	(character) The ID of the evaluation queue.
status	(Optional[character]) Optionally filter submissions by a submission status, such as SCORED, VALID, INVALID, OPEN, CLOSED or EVALUATION_IN_PROGRESS.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields [Submission](#) objects as they are retrieved from the API. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one [Submission](#) at a time.

Examples

```
## Not run:
## Getting submissions for an evaluation
# Get SCORED submissions from a specific evaluation.
library(synapser)
synLogin()

submissions = as.list(Submission() |> synGetEvaluationSubmissions(
  evaluation_id = "99999999",
  status = "SCORED"
))
print(sprintf("Found %d submissions", length(submissions)))

## End(Not run)
```

synGetFromId

File : from_id

Description

Wrapper for [synGet](#).

Usage

```
synGetFromId(cls, synapse_id, synapse_client=NULL)
```

Arguments

cls (File) The File class (or an existing File instance) to dispatch this call to.

synapse_id (str) The ID of the file in Synapse.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The file object.

Examples

```
## Not run:
## Using this function
# Assuming you have a file with the ID "syn123":
file_instance = File |> synGetFromId(synapse_id = "syn123")

## End(Not run)
```

synGetFromId	<i>Team : from_id</i>
--------------	-----------------------

Description

Gets Team object using its integer id.

Usage

```
synGetFromId(cls, id, synapse_client=NULL)
```

Arguments

cls	(Team) The Team class (or an existing Team instance) to dispatch this call to.
id	(integer) The id of the team.
synapse_client	(Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

Team: The Team object.

Examples

```
## Not run:  
## Get a team by its ID  
library(synapser)  
synLogin()  
  
team = Team() |> synGetFromId(id = 123456L)  
  
## End(Not run)
```

synGetFromId	<i>UserProfile : from_id</i>
--------------	------------------------------

Description

Gets UserProfile object using its integer id.

Usage

```
synGetFromId(cls, user_id, synapse_client=NULL)
```

Arguments

cls (UserProfile) The UserProfile class (or an existing UserProfile instance) to dispatch this call to.

user_id (int) The id of the user.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The UserProfile object.

Examples

```
## Not run:
## Using this function
# Assuming you know the integer ID of a Synapse user:
user_profile = UserProfile |> synGetFromId(user_id = 273948L)

## End(Not run)
```

synGetFromName	<i>Team : from_name</i>
----------------	-------------------------

Description

Gets Team object using its string name.

Note: You will be unable to retrieve a team by name immediately after its creation because the fragment service is eventually consistent. If you need to retrieve a team immediately following creation you should use the [synGetFromId\(\)](#) function instead.

Usage

```
synGetFromName(cls, name, synapse_client=NULL)
```

Arguments

cls (Team) The Team class (or an existing Team instance) to dispatch this call to.

name (character) The name of the team.

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

Team: The Team object.

Examples

```
## Not run:
## Get a team by its name
library(synapser)
synLogin()

team = Team() |> synGetFromName(name = "My Uniquely Named Team")

## End(Not run)
```

synGetFromParent	<i>Activity : from_parent</i>
------------------	-------------------------------

Description

Get the Activity from Synapse based on the parent entity.

Usage

```
synGetFromParent(cls, parent, parent_version_number=NULL, synapse_client=NULL)
```

Arguments

- cls (Activity) The Activity class (or an existing Activity instance) to dispatch this call to.
- parent (Union[Table, File, EntityView, Dataset, str]) The parent entity this activity is associated with. The parent may also have a version_number. Gets the most recent version if version is omitted.
- parent_version_number (Optional[int]) The version number of the parent entity. When parent is a string with version (e.g., "syn123.4"), the version in the string takes precedence. When parent is an object, this parameter takes precedence over parent\$version_number. Gets the most recent version if omitted.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The Activity object, or NULL if it does not exist.

Errors

An error is raised if the parent does not have an ID.

Examples

```
## Not run:
## Using this function
# Get the Activity associated with the latest version of an entity
library(synapser)
synLogin()

activity = Activity() |> synGetFromParent(parent = "syn123")
print(activity$name)

# Get the Activity associated with a specific version of an entity
activity = Activity() |> synGetFromParent(parent = "syn123", parent_version_number = 2L)

# Or include the version in the Synapse ID
activity = Activity() |> synGetFromParent(parent = "syn123.2")

## End(Not run)
```

synGetFromPath

File : from_path

Description

Get the file from Synapse. If the path of the file matches multiple files within Synapse the first one found will be returned. The other matching files will be printed to the console.

Wrapper for [synGet](#).

Usage

```
synGetFromPath(cls, path, synapse_client=NULL)
```

Arguments

cls	(File) The File class (or an existing File instance) to dispatch this call to.
path	(str) The path to the file on disk.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The file object.

Examples

```
## Not run:
## Using this function
# Assuming you have a file at the path "path/to/file.txt":
file_instance = File |> synGetFromPath(path = "path/to/file.txt")

## End(Not run)
```

synGetFromUsername	<i>UserProfile : from_username</i>
--------------------	------------------------------------

Description

Gets UserProfile object using its string name.

Usage

```
synGetFromUsername(cls, username, synapse_client=NULL)
```

Arguments

cls	(UserProfile) The UserProfile class (or an existing UserProfile instance) to dispatch this call to.
username	(str) A name chosen by the user that uniquely identifies them.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The UserProfile object.

Examples

```
## Not run:
## Using this function
# Assuming you know the username of a Synapse user:
user_profile = UserProfile |> synGetFromUsername(username = "myUsername")

## End(Not run)
```

`synGetInvalidValidation`*Folder : get_invalid_validation*

Description

Get invalid JSON schema validation results for a container entity.

Usage

```
synGetInvalidValidation(instance, synapse_client=NULL)
```

Arguments

`instance` (Folder) The Folder instance to operate on.

`synapse_client` (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields objects containing the validation response, all validation messages, and the validation exception details. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one result at a time.

Note

This only checks the direct children of this entity. It does not check this entity itself, and it does not check grandchildren (entities nested two or more levels below this entity).

Examples

```
## Not run:
## Using this function
# Retrieving invalid validation results for a folder. This example demonstrates
# how to get detailed invalid validation results for child entities within a container.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
```

```

SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create a file within the folder with invalid annotations to generate invalid results
invalid_file1 = File(
  path = FILE_PATH,
  parent_id = test_folder$id,
  annotations = list(foo = 123, bar = "not_an_integer") # both invalid
) |> synStore()
cat("Created file with invalid annotations\n")
Sys.sleep(2) # Allow time for processing

# Get invalid validation results
invalid_results = test_folder |> synGetInvalidValidation()
cat("Invalid validation items found:\n")
for (child in as.list(invalid_results)) {
  cat("Invalid validation item:\n")
  print(child)
}

## End(Not run)

```

synGetInvalidValidation

Project : get_invalid_validation

Description

Get invalid JSON schema validation results for the child entities of a container entity.

Usage

```
synGetInvalidValidation(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the validation response, all validation messages, and the validation exception details.

Note

This only checks the direct children of this entity. It does not check this entity itself, and it does not check grandchildren (entities nested two or more levels below this entity).

Examples

```
## Not run:
## Using this function
# Retrieving invalid validation results for a project. This example demonstrates
# how to get detailed invalid validation results for child entities within a container.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
```

```

my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to project:\n")
bound_schema

# Create a file within the project with invalid annotations to generate invalid results
invalid_file1 = File(
  path = FILE_PATH, # assumes you have something here or adjust path
  parent_id = PROJECT_ID
)
invalid_file1$annotations = list(foo = 123, bar = "not_an_integer") # both invalid
invalid_file1 = invalid_file1 |> synStore()
cat("Created file with invalid annotations\n")
Sys.sleep(2) # Allow time for processing

# Get invalid validation results
# synGetInvalidValidation() returns a Python generator, wrapped as a
# GeneratorWrapper -- materialize it with as.list() before iterating,
# since R's for() can't drive a Python generator directly.
invalid_results = as.list(test_project |> synGetInvalidValidation())
cat("Invalid validation items found:\n")
for (child in invalid_results) {
  cat(sprintf("Invalid validation item: %s\n", child))
}

## End(Not run)

```

Description

Gets the JSON Schemas that are part of this organization

Usage

```
synGetJsonSchemas(instance, synapse_client=NULL)
```

Arguments

`instance` (Organization) The Organization instance to operate on.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields the JSONSchema objects that belong to this organization. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one JSONSchema at a time.

Examples

```
## Not run:
## Get the JSONSchemas that are part of this Organization
library(synapser)
synLogin()

org = Organization(name = "dpetest")
js_generator = org |> synGetJsonSchemas()
for (item in as.list(js_generator)) {
  print(item)
}

## End(Not run)
```

synGetMarkdownFile	WikiPage : <i>get_markdown_file</i>
--------------------	-------------------------------------

Description

Get the markdown URL of this wiki page.

Usage

```
synGetMarkdownFile(instance, download_file=TRUE, download_location=NULL, synapse_client=NULL)
```

Arguments

instance (WikiPage) The WikiPage instance to operate on.

download_file (bool) Whether associated files should be downloaded. Default is TRUE.

download_location (Optional[str]) The directory to download the file to. Required if download_file is TRUE.

synapse_client (Optional[Synapse]) Optionally provide a Synapse client.

Value

If download_file is TRUE, the markdown file will be downloaded to the download_location. Otherwise, the URL will be returned.

Errors

An error is raised if owner_id or id is not provided.

Examples

```
## Not run:
## Example 1: Get the markdown URL for a wiki page
# This example shows how to get the markdown URL for a wiki page.
markdown_url = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetMarkdownFile(download_file = FALSE)
cat("Markdown URL:", markdown_url, "\n")

## Example 2: Download the markdown file for a wiki page
# This example shows how to download the markdown file for a wiki page.
markdown_file_path = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGetMarkdownFile(download_file = TRUE)
cat("Markdown file path:", markdown_file_path, "\n")

## End(Not run)
```

```
synGetOpenInvitations Team : open_invitations
```

Description

Gets all open invitations for a team given the ID field on the Team instance.

Usage

```
synGetOpenInvitations(instance, synapse_client=NULL)
```

Arguments

instance (Team) The Team instance to operate on.

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

List[dict]: A list of invitations.

Examples

```
## Not run:
## List the open invitations for a team
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
open_invitations = team |> synGetOpenInvitations()

## End(Not run)
```

synGetPermissions	<i>Dataset : get_permissions</i>
-------------------	----------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance (Dataset) The Dataset instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = Dataset(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>DatasetCollection : get_permissions</i>
-------------------	--

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = DatasetCollection(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>EntityView : get_permissions</i>
-------------------	-------------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

`instance` (EntityView) The EntityView instance to operate on.
`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = EntityView(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>Evaluation : get_permissions</i>
-------------------	-------------------------------------

Description

Get the user permissions for this evaluation.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

`instance` (Evaluation) The Evaluation instance to operate on.
`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A named list of the permissions the current user has on this evaluation.

Errors

- An error is raised if the evaluation's id is not set.
- An error is also raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Get permissions for the current user
library(synapser)
synLogin()

PROJECT_ID = "syn123456" # replace with your project ID

# Store a new evaluation in the project first
evaluation = Evaluation(
  name = "My Challenge Evaluation",
  description = "Evaluation for my data challenge",
  content_source = PROJECT_ID,
  submission_instructions_message = "Submit CSV files only",
  submission_receipt_message = "Thank you for your submission!"
) |> synStore()

# Get the permissions for the current user
my_permissions = evaluation |> synGetPermissions()
my_permissions

## End(Not run)
```

synGetPermissions	<i>File : get_permissions</i>
-------------------	-------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance	(File) The File instance to operate on.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = File(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>Folder : get_permissions</i>
-------------------	---------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = Folder(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>MaterializedView : get_permissions</i>
-------------------	---

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance	(MaterializedView) The MaterializedView instance to operate on.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = MaterializedView(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>Project : get_permissions</i>
-------------------	----------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = Project(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>SubmissionView : get_permissions</i>
-------------------	---

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance (SubmissionView) The SubmissionView instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = SubmissionView(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>Table : get_permissions</i>
-------------------	--------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance (Table) The Table instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = Table(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetPermissions	<i>VirtualTable : get_permissions</i>
-------------------	---------------------------------------

Description

Get the **permissions** that the caller has on an Entity.

Usage

```
synGetPermissions(instance, synapse_client=NULL)
```

Arguments

instance	(VirtualTable) The VirtualTable instance to operate on.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A Permissions object

Examples

```
## Not run:
## Using this function:
# Getting permissions for a Synapse Entity
library(synapser)
synLogin()

permissions = VirtualTable(id = "syn123") |> synGetPermissions()

# Getting access types list from the Permissions object
permissions$access_types

## End(Not run)
```

synGetProjectSetting	<i>Folder : get_project_setting</i>
----------------------	-------------------------------------

Description

Get the project setting for this entity.

Usage

```
synGetProjectSetting(instance, setting_type="upload", synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.

setting_type (character) The type of setting to retrieve. Currently only 'upload' is supported.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The ProjectSetting object, or NULL if no setting exists.

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Using this function
# Get the upload settings for a folder:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
setting = folder |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  print(sprintf("Storage locations: %s", paste(unlist(setting$locations), collapse = ", ")))
}

## End(Not run)
```

synGetProjectSetting *Project : get_project_setting*

Description

Get the project setting for this entity.

Usage

```
synGetProjectSetting(instance, setting_type="upload", synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

setting_type (character) The type of setting to retrieve. Currently only 'upload' is supported.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The ProjectSetting object, or NULL if no setting exists.

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Using this function
# Get the upload settings for a project:
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
setting = project |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  print(sprintf("Storage locations: %s", paste(unlist(setting$locations), collapse = ", ")))
}

## End(Not run)
```

synGetSchema	<i>EntityView : get_schema</i>
--------------	--------------------------------

Description

Get the JSON schema bound to the entity.

Usage

```
synGetSchema(instance, synapse_client=NULL)
```

Arguments

- instance (EntityView) The EntityView instance to operate on.
- synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the bound JSON schema.

Examples

```
## Not run:
## Using this function
# Retrieving the bound JSON schema from an EntityView. This example demonstrates
# how to get an existing schema binding from an entity that already has a
# schema bound. Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VIEW_NAME = "test_script_entityview"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test EntityView of the files in the project
test_view = EntityView(
  name = VIEW_NAME,
  parent_id = PROJECT_ID,
  scope_ids = list(PROJECT_ID),
  view_type_mask = 1L # 1 = files
) |> synStore()
cat("Created test EntityView:", VIEW_NAME, "\n")

# Bind JSON schema to the EntityView first
bound_schema = test_view |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to EntityView:\n")
bound_schema
```



```

# Retrieve the bound schema from the EntityView
bound_schema = test_view |> synGetSchema()
cat("Retrieved schema from EntityView:\n")
bound_schema

## End(Not run)

```

synGetSchema

*File : get_schema***Description**

Get the JSON schema bound to the entity.

Usage

```
synGetSchema(instance, synapse_client=NULL)
```

Arguments

instance (File) The File instance to operate on.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the bound JSON schema.

Examples

```

## Not run:
## Using this function
# Retrieving the bound JSON schema from a folder or file. This example demonstrates
# how to get existing schema bindings from entities that already have schemas bound.
# Set the PROJECT_NAME and FILE_PATH variables to your project name
# and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"

```

```

VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder first
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Retrieve the bound schema from the folder
bound_schema = test_folder |> synGetSchema()
cat("Retrieved schema from folder:\n")
bound_schema

# Retrieve the bound schema from the file
bound_schema_file = example_file |> synGetSchema()
cat("Retrieved schema from file:\n")
bound_schema_file

## End(Not run)

```

synGetSchema	<i>Folder : get_schema</i>
--------------	----------------------------

Description

Get the JSON schema bound to the entity.

Usage

```
synGetSchema(instance, synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the bound JSON schema.

Examples

```
## Not run:
## Using this function
# Retrieving the bound JSON schema from a folder or file. This example demonstrates
# how to get existing schema bindings from entities that already have schemas bound.
# Set the PROJECT_NAME and FILE_PATH variables to your project name
# and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
```

```

        type = "object",
        properties = list(
            foo = list(type = "string"),
            bar = list(type = "integer")
        ),
        required = list("foo")
    )
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
    json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder first
bound_schema = test_folder |> synBindSchema(
    json_schema_uri = SCHEMA_URI,
    enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
    json_schema_uri = SCHEMA_URI,
    enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Retrieve the bound schema from the folder
bound_schema = test_folder |> synGetSchema()
cat("Retrieved schema from folder:\n")
bound_schema

# Retrieve the bound schema from the file
bound_schema_file = example_file |> synGetSchema()
cat("Retrieved schema from file:\n")
bound_schema_file

## End(Not run)

```

synGetSchema

Project : get_schema

Description

Get the JSON schema bound to the entity.

Usage

```
synGetSchema(instance, synapse_client=NULL)
```

Arguments

`instance` (Project) The Project instance to operate on.

`synapse_client` (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the bound JSON schema.

Examples

```
## Not run:
## Using this function
# Retrieving the bound JSON schema from a project. This example demonstrates
# how to get an existing schema binding from an entity that already has a
# schema bound. Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)
```

```
# Bind JSON schema to the project first
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to project:\n")
bound_schema

# Retrieve the bound schema from the project
bound_schema = test_project |> synGetSchema()
cat("Retrieved schema from project:\n")
bound_schema

## End(Not run)
```

synGetSchema

Table : get_schema

Description

Get the JSON schema bound to the entity.

Usage

```
synGetSchema(instance, synapse_client=NULL)
```

Arguments

instance (Table) The Table instance to operate on.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing details about the bound JSON schema.

Examples

```
## Not run:
## Using this function
# Retrieving the bound JSON schema from a table. This example demonstrates
# how to get an existing schema binding from an entity that already has a
# schema bound. Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
```

```

PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
TABLE_NAME = "test_script_table"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test table
test_table = Table(name = TABLE_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test table:", TABLE_NAME, "\n")

# Bind JSON schema to the table first
bound_schema = test_table |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to table:\n")
bound_schema

# Retrieve the bound schema from the table
bound_schema = test_table |> synGetSchema()
cat("Retrieved schema from table:\n")
bound_schema

## End(Not run)

```

synGetSchemaDerivedKeys

EntityView : get_schema_derived_keys

Description

Retrieve derived JSON schema keys for the entity.

Usage

```
synGetSchemaDerivedKeys(instance, synapse_client=NULL)
```

Arguments

instance (EntityView) The EntityView instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the derived keys for the entity.

Examples

```
## Not run:
## Using this function
# Retrieving derived keys from an EntityView. This example demonstrates
# how to get derived annotation keys from schemas with constant values.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
DERIVED_TEST_SCHEMA_NAME = "myTestDerivedSchema" # replace with your derived schema name
VIEW_NAME = "test_script_entityview"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, DERIVED_TEST_SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"), # Example constant for derived annotation
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = DERIVED_TEST_SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)
```



```

# Create a test EntityView of the files in the project
test_view = EntityView(
    name = VIEW_NAME,
    parent_id = PROJECT_ID,
    scope_ids = list(PROJECT_ID),
    view_type_mask = 1L # 1 = files
) |> synStore()
cat("Created test EntityView:", VIEW_NAME, "\n")

# Bind JSON schema to the EntityView
bound_schema = test_view |> synBindSchema(
    json_schema_uri = SCHEMA_URI,
    enable_derived_annotations = TRUE
)
cat("Bound schema to EntityView with derived annotations:\n")
bound_schema

# Get the derived keys from the bound schema of the EntityView
test_view = test_view |> synSetProperties(
    annotations = list(foo = "test_value_new", bar = 42) # Example annotations
) |> synStore()
cat("Added annotations to EntityView and stored\n")

derived_keys = test_view |> synGetSchemaDerivedKeys()
cat("Derived keys from EntityView:\n")
derived_keys

## End(Not run)

```

synGetSchemaDerivedKeys

File : get_schema_derived_keys

Description

Retrieve derived JSON schema keys for the entity.

Usage

```
synGetSchemaDerivedKeys(instance, synapse_client=NULL)
```

Arguments

instance (File) The File instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the derived keys for the entity.

Examples

```
## Not run:
## Using this function
# Retrieving derived keys from a folder or file. This example demonstrates
# how to get derived annotation keys from schemas with constant values.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
DERIVED_TEST_SCHEMA_NAME = "myTestDerivedSchema" # replace with your derived schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, DERIVED_TEST_SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"), # Example constant for derived annotation
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = DERIVED_TEST_SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder with derived annotations:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()
```

```
bound_schema_file = example_file |> synBindSchema(  
  json_schema_uri = SCHEMA_URI,  
  enable_derived_annotations = TRUE  
)  
cat("Bound schema to file with derived annotations:\n")  
bound_schema_file  
  
# Get the derived keys from the bound schema of the folder  
test_folder = test_folder |> synSetProperties(  
  annotations = list(foo = "test_value_new", bar = 42L) # Example annotations  
) |> synStore()  
cat("Added annotations to folder and stored\n")  
  
derived_keys = test_folder |> synGetSchemaDerivedKeys()  
cat("Derived keys from folder:\n")  
derived_keys  
  
# Get the derived keys from the bound schema of the file  
example_file = example_file |> synSetProperties(  
  annotations = list(foo = "test_value_new", bar = 43L) # Example annotations  
) |> synStore()  
cat("Added annotations to file and stored\n")  
  
derived_keys_file = example_file |> synGetSchemaDerivedKeys()  
cat("Derived keys from file:\n")  
derived_keys_file  
  
## End(Not run)
```

synGetSchemaDerivedKeys

Folder : get_schema_derived_keys

Description

Retrieve derived JSON schema keys for the entity.

Usage

```
synGetSchemaDerivedKeys(instance, synapse_client=NULL)
```

Arguments

instance (Folder) The Folder instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the derived keys for the entity.

Examples

```
## Not run:
## Using this function
# Retrieving derived keys from a folder or file. This example demonstrates
# how to get derived annotation keys from schemas with constant values.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
DERIVED_TEST_SCHEMA_NAME = "myTestDerivedSchema" # replace with your derived schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, DERIVED_TEST_SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"), # Example constant for derived annotation
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = DERIVED_TEST_SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder with derived annotations:\n")
```

```

bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file with derived annotations:\n")
bound_schema_file

# Get the derived keys from the bound schema of the folder
test_folder = test_folder |> synSetProperties(
  annotations = list(foo = "test_value_new", bar = 42L) # Example annotations
) |> synStore()
cat("Added annotations to folder and stored\n")

derived_keys = test_folder |> synGetSchemaDerivedKeys()
cat("Derived keys from folder:\n")
derived_keys

# Get the derived keys from the bound schema of the file
example_file = example_file |> synSetProperties(
  annotations = list(foo = "test_value_new", bar = 43L) # Example annotations
) |> synStore()
cat("Added annotations to file and stored\n")

derived_keys_file = example_file |> synGetSchemaDerivedKeys()
cat("Derived keys from file:\n")
derived_keys_file

## End(Not run)

```

synGetSchemaDerivedKeys

Project : get_schema_derived_keys

Description

Retrieve derived JSON schema keys for the entity.

Usage

```
synGetSchemaDerivedKeys(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the derived keys for the entity.

Examples

```
## Not run:
## Using this function
# Retrieving derived keys from a project. This example demonstrates
# how to get derived annotation keys from schemas with constant values.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
DERIVED_TEST_SCHEMA_NAME = "myTestDerivedSchema" # replace with your derived schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, DERIVED_TEST_SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"), # Example constant for derived annotation
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = DERIVED_TEST_SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to project with derived annotations:\n")
bound_schema

# Get the derived keys from the bound schema of the project
```

```

test_project = test_project |> synSetProperties(
  annotations = list(foo = "test_value_new", bar = 42L) # Example annotations
) |> synStore()
cat("Added annotations to project and stored\n")

derived_keys = test_project |> synGetSchemaDerivedKeys()
cat("Derived keys from project:\n")
derived_keys

## End(Not run)

```

synGetSchemaDerivedKeys

Table : get_schema_derived_keys

Description

Retrieve derived JSON schema keys for the entity.

Usage

```
synGetSchemaDerivedKeys(instance, synapse_client=NULL)
```

Arguments

instance (Table) The Table instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

An object containing the derived keys for the entity.

Examples

```

## Not run:
## Using this function
# Retrieving derived keys from a table. This example demonstrates
# how to get derived annotation keys from schemas with constant values.
# Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name

```

```

DERIVED_TEST_SCHEMA_NAME = "myTestDerivedSchema" # replace with your derived schema name
TABLE_NAME = "test_script_table"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, DERIVED_TEST_SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"), # Example constant for derived annotation
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = DERIVED_TEST_SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test table
test_table = Table(name = TABLE_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test table:", TABLE_NAME, "\n")

# Bind JSON schema to the table
bound_schema = test_table |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to table with derived annotations:\n")
bound_schema

# Get the derived keys from the bound schema of the table
test_table = test_table |> synSetProperties(
  annotations = list(foo = "test_value_new", bar = 42) # Example annotations
) |> synStore()
cat("Added annotations to table and stored\n")

derived_keys = test_table |> synGetSchemaDerivedKeys()
cat("Derived keys from table:\n")
derived_keys

## End(Not run)

```

synGetSchemaValidationStatistics

Folder : get_schema_validation_statistics

Description

Get validation statistics for a container entity.

Usage

```
synGetSchemaValidationStatistics(instance, synapse_client=NULL)
```

Arguments

`instance` (Folder) The Folder instance to operate on.

`synapse_client` (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation statistics.

Examples

```
## Not run:
## Using this function
# Retrieving validation statistics for a folder. This example demonstrates
# how to get validation statistics for a container entity after creating
# entities with various validation states. Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
```

```

)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create a file within the folder with invalid annotations to generate statistics
invalid_file1 = File(
  path = FILE_PATH,
  parent_id = test_folder$id,
  annotations = list(foo = 123, bar = "not_an_integer") # both invalid
) |> synStore()
cat("Created file with invalid annotations\n")
Sys.sleep(2) # Allow time for processing

stats = test_folder |> synGetSchemaValidationStatistics()
cat("Validation statistics:\n")
stats

## End(Not run)

```

```
synGetSchemaValidationStatistics
```

Project : get_schema_validation_statistics

Description

Get validation statistics for a container entity.

Usage

```
synGetSchemaValidationStatistics(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation statistics.

Examples

```
## Not run:
## Using this function
# Retrieving validation statistics for a project. This example demonstrates
# how to get validation statistics for a container entity after creating
# entities with various validation states. Set the PROJECT_NAME variable to your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    baz = list(type = "string", const = "example_value"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to project:\n")
bound_schema
```

```

# Create a file within the project with invalid annotations to generate statistics
invalid_file = File(
  path = FILE_PATH, # assumes you have something here or adjust path
  parent_id = PROJECT_ID
)
invalid_file$annotations = list(foo = 123, bar = "not_an_integer") # both invalid
invalid_file = invalid_file |> synStore()
cat("Created file with invalid annotations\n")
Sys.sleep(2) # Allow time for processing

stats = test_project |> synGetSchemaValidationStatistics()
cat("Validation statistics:\n")
stats

## End(Not run)

```

synGetSession

Agent : *get_session*

Description

Gets an existing agent session. Adds the session to the Agent's sessions named list and sets it as the current session.

Usage

```
synGetSession(instance, session_id, synapse_client=NULL)
```

Arguments

instance (Agent) The Agent instance to operate on.

session_id (str) The ID of the session to get.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The existing AgentSession object.

Examples

```

## Not run:
## Get an existing session and send a prompt.
# Retrieve an existing session by providing the session ID and calling synGetSession().
# Then, send a prompt to the agent.
library(synapser)
synLogin()

my_session = Agent() |> synGetSession(session_id = "foo")

```

```

my_session |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## End(Not run)

```

synGetStsStorageToken *Folder : get_sts_storage_token*

Description

Get STS (AWS Security Token Service) credentials for direct access to the storage location backing this entity. These credentials can be used with AWS tools such as the AWS CLI.

Usage

```
synGetStsStorageToken(instance, permission, output_format="json", min_remaining_life=NULL, synapse_client)
```

Arguments

instance	(Folder) The Folder instance to operate on.
permission	(character) The permission level for the token. Must be 'read_only' or 'read_write'.
output_format	(character) The output format for the credentials. Options: <ul style="list-style-type: none"> - 'json' (default): the credentials exactly as returned by Synapse, as a named list. - 'boto': a named list with only the aws_access_key_id, aws_secret_access_key, and aws_session_token entries. - 'bash', 'cmd', or 'powershell': a character string of commands that export the credentials as environment variables in that shell. - 'shell': the same as one of 'bash', 'cmd', or 'powershell', chosen automatically for the current system.
min_remaining_life	(Optional[integer]) The minimum remaining life (in seconds) for a cached token before a new one is fetched.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The STS credentials in the requested format.

Errors

Raises an error if the entity does not have an id set.

Note

The entity must use a storage location that has STS enabled.

Examples

```
## Not run:
## Using credentials with AWS tools
# Get STS credentials for an STS-enabled folder and make them available to
# AWS tools (such as the AWS CLI) through the standard environment variables.
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
credentials = folder |> synGetStsStorageToken(
  permission = "read_write",
  output_format = "boto"
)
Sys.setenv(
  AWS_ACCESS_KEY_ID = credentials$aws_access_key_id,
  AWS_SECRET_ACCESS_KEY = credentials$aws_secret_access_key,
  AWS_SESSION_TOKEN = credentials$aws_session_token
)

## End(Not run)
```

synGetStsStorageToken *Project : get_sts_storage_token*

Description

Get STS (AWS Security Token Service) credentials for direct access to the storage location backing this entity. These credentials can be used with AWS tools such as the AWS CLI.

Usage

```
synGetStsStorageToken(instance, permission, output_format="json", min_remaining_life=NULL, synapse_c)
```

Arguments

instance	(Project) The Project instance to operate on.
permission	(character) The permission level for the token. Must be 'read_only' or 'read_write'.
output_format	(character) The output format for the credentials. Options: <ul style="list-style-type: none"> - 'json' (default): the credentials exactly as returned by Synapse, as a named list. - 'boto': a named list with only the aws_access_key_id, aws_secret_access_key, and aws_session_token entries. - 'bash', 'cmd', or 'powershell': a character string of commands that export the

credentials as environment variables in that shell.

- 'shell': the same as one of 'bash', 'cmd', or 'powershell', chosen automatically for the current system.

min_remaining_life

(Optional[integer]) The minimum remaining life (in seconds) for a cached token before a new one is fetched.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The STS credentials in the requested format.

Errors

Raises an error if the entity does not have an id set.

Note

The entity must use a storage location that has STS enabled.

Examples

```
## Not run:
## Using credentials with AWS tools
# Get STS credentials for an STS-enabled project and make them available to
# AWS tools (such as the AWS CLI) through the standard environment variables.
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
credentials = project |> synGetStsStorageToken(
  permission = "read_write",
  output_format = "boto"
)
Sys.setenv(
  AWS_ACCESS_KEY_ID = credentials$aws_access_key_id,
  AWS_SECRET_ACCESS_KEY = credentials$aws_secret_access_key,
  AWS_SESSION_TOKEN = credentials$aws_session_token
)

## End(Not run)
```

synGetSubmissionCount *Submission : get_submission_count*

Description

Gets the number of Submissions for a specified Evaluation queue.

Usage

```
synGetSubmissionCount(evaluation_id, synapse_client=NULL)
```

Arguments

`evaluation_id` (character) The ID of the evaluation queue.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The API response containing the submission count.

Examples

```
## Not run:
## Getting submission count
# Get the total number of submissions from a specific evaluation.
library(synapser)
synLogin()

response = synGetSubmissionCount(evaluation_id = "9999999")
print(response)

## End(Not run)
```

synGetTeamMembers *Team : members*

Description

Gets the TeamMembers associated with a team given the ID field on the Team instance.

Usage

```
synGetTeamMembers(instance, synapse_client=NULL)
```


Arguments

instance (Team) The Team instance to operate on.

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

List[TeamMember]: A List of TeamMember objects.

Examples

```
## Not run:
## List the members of a team
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
members = team |> synGetTeamMembers()
for (member in members) {
  cat(sprintf("%s (admin: %s)\n", member$member$user_name, member$is_admin))
}

## End(Not run)
```

synGetUserMembershipStatus

Team : get_user_membership_status

Description

Retrieve a user's Team Membership Status bundle for this team.

Usage

```
synGetUserMembershipStatus(instance, user_id, synapse_client=NULL)
```

Arguments

instance (Team) The Team instance to operate on.

user_id (character) Synapse user ID

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

TeamMembershipStatus object

Examples

```
## Not run:
## Check if a user is a member of a team
# This example shows how to check a user's membership status in a team.
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)

# Check membership status for a specific user
user_id = "3350396" # Replace with actual user ID
status = team |> synGetUserMembershipStatus(user_id = user_id)

cat("User ID:", status$user_id, "\n")
cat("Is member:", status$is_member, "\n")
cat("Can join:", status$can_join, "\n")
cat("Has open invitation:", status$has_open_invitation, "\n")
cat("Has open request:", status$has_open_request, "\n")
cat("Membership approval required:", status$membership_approval_required, "\n")

## End(Not run)
```

synGetUserSubmissionBundles

SubmissionBundle : get_user_submission_bundles

Description

Retrieves all user bundled Submissions and SubmissionStatuses for a specified Evaluation.

Usage

```
synGetUserSubmissionBundles(cls, evaluation_id, synapse_client=NULL)
```

Arguments

cls (SubmissionBundle) The SubmissionBundle class (or an existing Submission-Bundle instance) to dispatch this call to.

evaluation_id (character) The ID of the specified Evaluation.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields [SubmissionBundle](#) objects as they are retrieved from the API. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one [SubmissionBundle](#) at a time.

Examples

```
## Not run:
## Getting user submission bundles
library(synapser)
synLogin()

bundles = as.list(SubmissionBundle() |> synGetUserSubmissionBundles(
  evaluation_id = "9999999"
))
print(sprintf("Found %d user submission bundles", length(bundles)))
for (bundle in bundles) {
  print(sprintf("Submission ID: %s", bundle$submission$id))
}

## End(Not run)
```

synGetUserSubmissions *Submission* : *get_user_submissions*

Description

Retrieves all user Submissions for a specified Evaluation queue.

Usage

```
synGetUserSubmissions(cls, evaluation_id, synapse_client=NULL)
```

Arguments

cls (Submission) The Submission class (or an existing Submission instance) to dispatch this call to.

evaluation_id (character) The ID of the evaluation queue.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields [Submission](#) objects as they are retrieved from the API. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one [Submission](#) at a time.

Examples

```
## Not run:
## Getting user submissions
library(synapser)
synLogin()
```

```

    submissions = as.list(Submission() |> synGetUserSubmissions(
        evaluation_id = "99999999"
    ))
    print(sprintf("Found %d user submissions", length(submissions)))

## End(Not run)

```

synIndexFilesForMigration

Folder : index_files_for_migration

Description

Index files in this entity for migration to a new storage location.

This is the first step in migrating files to a new storage location. After indexing, use [synMigrateIndexedFiles\(\)](#) to perform the actual migration.

Usage

```
synIndexFilesForMigration(instance, dest_storage_location_id, db_path=NULL, source_storage_location_
```

Arguments

instance	(Folder) The Folder instance to operate on.
dest_storage_location_id	(integer) The destination storage location ID.
db_path	(Optional[character]) Path to the SQLite database file for tracking migration state. If not provided, a temporary directory will be used. The path can be retrieved from the db_path field of the returned MigrationResult.
source_storage_location_ids	(Optional[list[integer]]) Optional list of source storage location IDs to filter which files to migrate. If NULL, all files are indexed.
file_version_strategy	(character) Strategy for handling file versions. Options: 'new' (default) - create new versions, 'all' - migrate all versions, 'latest' - only migrate latest version, 'skip' - skip if file exists.
include_table_files	(logical) Whether to include files attached to tables.
continue_on_error	(logical) Whether to continue indexing if an error occurs.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A MigrationResult object containing indexing statistics and the database path (accessible via result\$db_path).

Examples

```
## Not run:
## Indexing files for migration
# Index files in a folder for migration:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
result = folder |> synIndexFilesForMigration(
  dest_storage_location_id = 12345L
)
print(sprintf("Database path: %s", result$db_path))
print(result$counts_by_status)

## End(Not run)
```

```
synIndexFilesForMigration
```

```
Project : index_files_for_migration
```

Description

Index files in this entity for migration to a new storage location.

This is the first step in migrating files to a new storage location. After indexing, use [synMigrateIndexedFiles\(\)](#) to perform the actual migration.

Usage

```
synIndexFilesForMigration(instance, dest_storage_location_id, db_path=NULL, source_storage_location_
```

Arguments

instance	(Project) The Project instance to operate on.
dest_storage_location_id	(integer) The destination storage location ID.
db_path	(Optional[character]) Path to the SQLite database file for tracking migration state. If not provided, a temporary directory will be used. The path can be retrieved from the db_path field of the returned MigrationResult.
source_storage_location_ids	(Optional[list[integer]]) Optional list of source storage location IDs to filter which files to migrate. If NULL, all files are indexed.
file_version_strategy	(character) Strategy for handling file versions. Options: 'new' (default) - create new versions, 'all' - migrate all versions, 'latest' - only migrate latest version, 'skip' - skip if file exists.
include_table_files	(logical) Whether to include files attached to tables.

continue_on_error (logical) Whether to continue indexing if an error occurs.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A MigrationResult object containing indexing statistics and the database path (accessible via result\$db_path).

Examples

```
## Not run:
## Indexing files for migration
# Index files in a project for migration:
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
result = project |> synIndexFilesForMigration(
  dest_storage_location_id = 12345L
)
print(sprintf("Database path: %s", result$db_path))
print(result$counts_by_status)

## End(Not run)
```

synInviteToTeam	<i>Team : invite</i>
-----------------	----------------------

Description

Invites a user to a team given the ID field on the Team instance.

Usage

```
synInviteToTeam(instance, user, message, force=TRUE, synapse_client=NULL)
```

Arguments

instance (Team) The Team instance to operate on.

user The username or ID of the user to invite.

message (character) The message to send.

force (logical) If TRUE, will send the invite even if the user is already a member or has an open invitation. If FALSE, will not send the invite if the user is already a member or has an open invitation. Defaults to TRUE.

synapse_client (Optional[Synapse]) If not provided, the last created instance from the Synapse class constructor will be used.

Value

The invite response or NULL if an invite was not sent.

Examples

```
## Not run:
## Invite a user to a team
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
team |> synInviteToTeam(user = "my_username", message = "Please join my team!")

## End(Not run)
```

synIsCertified	<i>UserProfile : is_certified</i>
----------------	-----------------------------------

Description

Determine whether a user is certified.

Usage

```
synIsCertified(instance, synapse_client=NULL)
```

Arguments

instance (UserProfile) The UserProfile instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

TRUE if the user is certified, FALSE otherwise.

Errors

An error is raised if neither id nor username is specified.

Examples

```
## Not run:
## Using this function
# Check whether a user is certified
is_certified = UserProfile(username = "myUsername") |> synIsCertified()

## End(Not run)
```

synIsSynapseId	<i>synIsSynapseId</i>
----------------	-----------------------

Description

Check if given synID is valid (attached to actual entity).

Usage

```
synIsSynapseId(syn_id, synapse_client=NULL)
```

Arguments

syn_id (str) A Synapse ID to validate.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

TRUE if the Synapse ID is valid.

Examples

```
## Not run:
## Example 1: Validating a Synapse ID
# Check if a Synapse ID exists:
library(synapser)
synLogin()

if (synIsSynapseId("syn123456")) {
  print("Valid Synapse ID")
} else {
  print("Invalid or non-existent Synapse ID")
}

## Example 2: Validating multiple IDs
# Check multiple Synapse IDs in a loop:
library(synapser)
synLogin()

ids_to_check = list("syn123456", "syn999999", "syn789012")
for (synapse_id in ids_to_check) {
  if (synIsSynapseId(synapse_id)) {
    print(sprintf("%s is valid", synapse_id))
  } else {
    print(sprintf("%s is invalid or does not exist", synapse_id))
  }
}
```



```
## Example 3: Using in conditional logic
# Use ID validation before attempting operations:
library(synpser)
synLogin()

user_input = "syn123456"
if (synIsSynapseId(user_input)) {
  entity = synGet(synapse_id = user_input)
  print(sprintf("Retrieved entity: %s", entity$name))
} else {
  print("Please provide a valid Synapse ID")
}

## End(Not run)
```

synList

Column : list

Description

List columns with optional prefix filtering.

Usage

```
synList(prefix=NULL, limit=100L, offset=0L, synapse_client=NULL)
```

Arguments

prefix	(Optional[str]) Optional prefix to filter columns by name.
limit	(int) Number of columns to retrieve per request to Synapse (pagination parameter). The function will continue retrieving results until all matching columns are returned.
offset	(int) The index of the first column to return (pagination parameter).
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A GeneratorWrapper object that yields [Column](#) instances. Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one [Column](#) instance at a time.

Examples

```
## Not run:
## Example 1: Getting all columns
# This example shows how you may list all columns.
library(synpser)
synLogin()
```

```

columns = synList()
for (column in as.list(columns)) {
  print(column$name)
}

## Example 2: Getting columns with a prefix
# This example shows how you may list columns filtered by a name prefix.
library(synapser)
synLogin()

columns = synList(prefix = "my_prefix")
for (column in as.list(columns)) {
  print(column$name)
}

## End(Not run)

```

synListAcl

Dataset : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
recursive	(logical) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(logical) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.

target_entity_types	(Optional[list[character]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(logical) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- all_entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = Dataset(id = "syn123") |> synListAcl()
print(acl_result)

# Access entity ACLs (all_entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        print(sprintf("Principal 273948 has permissions: %s", paste(unlist(acl_entry$permissions), collapse=" ")))
      }
    }
  }
}

# I can also access the ACL for the dataset itself
print(acl_result$entity_acl)

print(acl_result)
```

```
## Example 2: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs. If the dataset has no local sharing settings,
# there are no ACLs to display and no tree is produced.
library(synapser)
synLogin()

acl_result = Dataset(id = "syn123") |> synListAcl(
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
print(acl_result$ascii_tree)

## End(Not run)
```

synListAcl

DatasetCollection : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
recursive	(logical) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(logical) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.

target_entity_types	(Optional[list[character]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(logical) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- all_entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = DatasetCollection(id = "syn123") |> synListAcl()
print(acl_result)

# Access entity ACLs (all_entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        print(sprintf("Principal 273948 has permissions: %s", paste(unlist(acl_entry$permissions), collapse=" ")))
      }
    }
  }
}

# I can also access the ACL for the dataset collection itself
print(acl_result$entity_acl)

print(acl_result)
```

```

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (all_entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  print(sprintf("Entity %s has ACL with %d principals", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
print(acl_result$entity_acl)

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
print(acl_result$ascii_tree)

## End(Not run)

```

synListAcl

EntityView : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```

## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = EntityView(id = "syn123") |> synListAcl()
acl_result

# I can also access the ACL for the EntityView itself
acl_result$entity_acl

acl_result

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (all_entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.

library(synapser)
synLogin()

```



```

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)

```

synListAcl

File : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log=FALSE)
```

Arguments

instance	(File) The File instance to operate on.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").

`log_tree` (bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An `AclListResult` object containing a structured representation of ACLs where:

- `entity_acls`: A list of `EntityAcl` objects, each representing one entity's ACL
- Each `EntityAcl` contains `acl_entries` (a list of `AclEntry` objects)
- Each `AclEntry` contains the `principal_id` and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = File(id = "syn123") |> synListAcl()
acl_result

# Access entity ACLs (entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        cat(sprintf("Principal 273948 has permissions: %s\n", acl_entry$permissions))
      }
    }
  }
}

# I can also access the ACL for the file itself
acl_result$entity_acl

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
```

```

)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)

```

synListAcl

Folder : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

instance	(Folder) The Folder instance to operate on.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```

## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl()
acl_result

# Access entity ACLs (entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        cat(sprintf("Principal 273948 has permissions: %s\n", acl_entry$permissions))
      }
    }
  }
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,

```

```
target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)
```

synListAcl	<i>MaterializedView : list_acl</i>
------------	------------------------------------

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log.
```

Arguments

- instance (MaterializedView) The MaterializedView instance to operate on.
- recursive (bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the [synSyncFromSynapse\(\)](#) function.

include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = MaterializedView(id = "syn123") |> synListAcl()
acl_result

# Access entity ACLs (entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        cat(sprintf("Principal 273948 has permissions: %s\n", acl_entry$permissions))
      }
    }
  }
}
```

```

# I can also access the ACL for the file itself
acl_result$entity_acl

acl_result

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

```



```
## End(Not run)
```

```
synListAcl
```

```
Project : list_acl
```

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

instance	(Project) The Project instance to operate on.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = Project(id = "syn123") |> synListAcl()
acl_result

# Access entity ACLs (entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
    for (acl_entry in entity_acl$acl_entries) {
      if (acl_entry$principal_id == "273948") {
        cat(sprintf("Principal 273948 has permissions: %s\n", acl_entry$permissions))
      }
    }
  }
}

# I can also access the ACL for the file itself
acl_result$entity_acl

acl_result

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
```

```

        include_container_content = TRUE,
        target_entity_types = list("folder")
    )

    # List ACLs for specific entity types (e.g., tables and views)
    table_view_acl_result = Folder(id = "syn123") |> synListAcl(
        recursive = TRUE,
        include_container_content = TRUE,
        target_entity_types = list("table", "entityview", "materializedview")
    )

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
    recursive = TRUE,
    include_container_content = TRUE,
    log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)

```

synListAcl

SubmissionView : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = SubmissionView(id = "syn123") |> synListAcl()
acl_result

# Access entity ACLs (entity_acls is a list, not a named list)
for (entity_acl in acl_result$all_entity_acls) {
  if (entity_acl$entity_id == "syn123") {
    # Access individual ACL entries
```

```

        for (acl_entry in entity_acl$acl_entries) {
          if (acl_entry$principal_id == "273948") {
            cat(sprintf("Principal 273948 has permissions: %s\n", acl_entry$permissions))
          }
        }
      }
    }

    # I can also access the ACL for the file itself
    acl_result$entity_acl

    acl_result

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,

```

```
        include_container_content = TRUE,
        log_tree = TRUE # Enable ASCII tree logging
    )

    # The ASCII tree representation of the ACLs will also be available
    # in acl_result$ascii_tree
    acl_result$ascii_tree

## End(Not run)
```

synListAcl	Table : list_acl
------------	------------------

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log,
```

Arguments

- instance (Table) The Table instance to operate on.
- recursive (bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the [synSyncFromSynapse\(\)](#) function.
- include_container_content (bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
- target_entity_types (Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
- log_tree (bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = Table(id = "syn123") |> synListAcl()
acl_result

# I can also access the ACL for the file itself
acl_result$entity_acl

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)
```

```

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)

```

synListAcl

VirtualTable : list_acl

Description

List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

Usage

```
synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log
```

Arguments

instance (VirtualTable) The VirtualTable instance to operate on.

recursive	(bool) If TRUE and the entity is a container (e.g., Project or Folder), recursively process child containers. Note that this must be used with include_container_content=TRUE to have any effect. Setting recursive=TRUE with include_container_content=FALSE will raise an error. Only works on classes that support the synSyncFromSynapse() function.
include_container_content	(bool) If TRUE, include ACLs from contents directly within containers (files and folders inside self). This must be set to TRUE for recursive to have any effect. Defaults to FALSE.
target_entity_types	(Optional[list[str]]) Specify which entity types to process when listing ACLs. Allowed values are "folder", "file", "project", "table", "entityview", "materializedview", "virtualtable", "dataset", "datasetcollection", "submissionview" (case-insensitive). If NULL, defaults to list("folder", "file").
log_tree	(bool) If TRUE, logs the ACL results to console in ASCII tree format showing entity hierarchies and their ACL permissions in a tree-like structure. Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An AclListResult object containing a structured representation of ACLs where:

- entity_acls: A list of EntityAcl objects, each representing one entity's ACL
- Each EntityAcl contains acl_entries (a list of AclEntry objects)
- Each AclEntry contains the principal_id and their list of permissions

Errors

- An error is raised if the entity does not have an ID or if an invalid entity type is provided.
- An error is also raised if there are permission issues accessing ACLs.
- An error may also be raised for any other issues that occur during the process.

Examples

```
## Not run:
## Example 1: List ACLs for a single entity
library(synapser)
synLogin()

acl_result = VirtualTable(id = "syn123") |> synListAcl()
acl_result

# I can also access the ACL for the file itself
acl_result$entity_acl

## Example 2: List ACLs recursively for a folder and all its children
library(synapser)
synLogin()
```

```

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE
)

# Access each entity's ACL (entity_acls is a list)
for (entity_acl in acl_result$all_entity_acls) {
  cat(sprintf("Entity %s has ACL with %d principals\n", entity_acl$entity_id, length(entity_acl$acl_entries)))
}

# I can also access the ACL for the folder itself
acl_result$entity_acl

# List ACLs for only folder entities
folder_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("folder")
)

# List ACLs for specific entity types (e.g., tables and views)
table_view_acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  target_entity_types = list("table", "entityview", "materializedview")
)

## Example 3: List ACLs with ASCII tree visualization
# When log_tree = TRUE, the ACLs will be logged in a tree format. Additionally,
# the ascii_tree attribute of the AclListResult will contain the ASCII tree
# representation of the ACLs.
library(synapser)
synLogin()

acl_result = Folder(id = "syn123") |> synListAcl(
  recursive = TRUE,
  include_container_content = TRUE,
  log_tree = TRUE # Enable ASCII tree logging
)

# The ASCII tree representation of the ACLs will also be available
# in acl_result$ascii_tree
acl_result$ascii_tree

## End(Not run)

```

Description

Valid combinations of synLogin() arguments:

- authToken
- Profile-based authentication (from .synapseConfig)

If no login arguments are provided or only username is provided, synLogin() will attempt to log in using information from these sources (in order of preference):

1. .synapseConfig file (supports multiple profiles)(in user home folder unless configured otherwise)
2. User defined arguments during a CLI session
3. User's Personal Access Token (aka: Synapse Auth Token) from the environment variable: SYNAPSE_AUTH_TOKEN
4. Retrieves user's authentication token from AWS SSM Parameter store (if configured)

Usage

```
synLogin(email=NULL, silent=FALSE, authToken=NULL, profile="default")
```

Arguments

email	(str) Synapse user name (or an email address associated with a Synapse account)
silent	(bool) Defaults to FALSE. Suppresses the "Welcome ...!" message.
authToken	(str) A bearer authorization token, e.g. a personal access token .
profile	(str) Profile to use from .synapseConfig (default: "default").

Value

NULL

Examples

```
## Not run:
## Logging in
library(synapser)

# Logging in using a specific profile:
synLogin(profile = "user1")
# Prints: Welcome, username! You are using the user1 profile

# Logging in with an authentication token:
synLogin(authToken = "your_auth_token")
# Prints: Welcome, username!

## End(Not run)
```

synLogout

synLogout

Description

Removes authentication information from the Synapse client.

Usage

```
synLogout()
```

Value

NULL

Examples

```
## Not run:
## Logging out
library(synapser)
synLogin()

# ... work with Synapse ...

synLogout()

## End(Not run)
```

synMapDirectoryToAllContainedFiles

Folder : map_directory_to_all_contained_files

Description

Recursively loop over all of the already retrieved files and folders. Then return back a named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

This is used during the creation of the manifest TSV file during the syncFromSynapse utility function.

Usage

```
synMapDirectoryToAllContainedFiles(instance, root_path)
```

Arguments

instance (Folder) The Folder instance to operate on.
root_path (str) The root path where the top level files are stored.

Value

A named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

Examples

```
## Not run:  
## Using this function  
# Returning back a named list with 2 keys:  
#  
# Given:  
# root_folder  
# |-- sub_folder  
# |   |-- file1.txt  
# |   `-- file2.txt  
#  
# Returns:  
# list(  
#   "root_folder" = list(file1, file2),  
#   "root_folder/sub_folder" = list(file1, file2)  
# )  
  
# Returning back a named list with 3 keys:  
#  
# Given:  
# root_folder  
# |-- sub_folder_1  
# |   |-- file1.txt  
# |-- sub_folder_2  
# |   `-- file2.txt  
#  
# Returns:  
# list(  
#   "root_folder" = list(file1, file2),  
#   "root_folder/sub_folder_1" = list(file1),  
#   "root_folder/sub_folder_2" = list(file2)  
# )  
  
## End(Not run)
```

Description

Recursively loop over all of the already retrieved files and folders. Then return back a named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

This is used during the creation of the manifest TSV file during the syncFromSynapse utility function.

Usage

```
synMapDirectoryToAllContainedFiles(instance, root_path)
```

Arguments

instance	(Project) The Project instance to operate on.
root_path	(str) The root path where the top level files are stored.

Value

A named list where the key is the path to the directory at each level. The value is a list of all files in that directory AND all files in the child directories.

Examples

```
## Not run:
## Using this function
# Returning back a named list with 2 keys:
#
#   Given:
#   root_folder
#   |-- sub_folder
#   |   |-- file1.txt
#   |   |-- file2.txt
#
#   Returns:
#   list(
#     root_folder = list(file1, file2),
#     "root_folder/sub_folder" = list(file1, file2)
#   )
#
#
# Returning back a named list with 3 keys:
#
#   Given:
#   root_folder
#   |-- sub_folder_1
#   |   |-- file1.txt
#   |-- sub_folder_2
#   |   |-- file2.txt
#
#   Returns:
```

```

#    list(
#        root_folder = list(file1, file2),
#        "root_folder/sub_folder_1" = list(file1),
#        "root_folder/sub_folder_2" = list(file2)
#    )

## End(Not run)

```

synMd5Query

synMd5Query

Description

Find the Entities which have attached file(s) which have the given MD5 hash.

Usage

```
synMd5Query(md5, synapse_client=NULL)
```

Arguments

md5 (str) The MD5 hash to query for (hexadecimal string).

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of Entity headers matching the MD5 hash.

Examples

```

## Not run:
## Example 1: Finding entities by MD5 hash
# Search for entities with a specific MD5 hash:
library(synapser)
synLogin()

md5_hash = "1234567890abcdef1234567890abcdef"
results = synMd5Query(md5_hash)

print(sprintf("Found %d entities with MD5: %s", length(results), md5_hash))
for (entity in results) {
    print(sprintf("- %s: %s", entity$id, entity$name))
}

## Example 2: Checking for duplicate files
# Use MD5 query to find duplicate files before uploading:
library(synapser)
synLogin()

```

```

# Calculate MD5 of a local file
file_path = "/path/to/file.txt"
file_md5 = unname(tools::md5sum(file_path))

# Check if file already exists in Synapse
existing_entities = synMd5Query(file_md5)
if (length(existing_entities) > 0) {
  print("File already exists in Synapse:")
  for (entity in existing_entities) {
    print(sprintf("- %s: %s", entity$id, entity$name))
  }
} else {
  print("File is unique, safe to upload")
}

## Example 3: Finding all versions of a file
# Find all entities that share the same content (MD5):
library(synapser)
synLogin()

# Get MD5 from a known file
file = synGet(synapse_id = "syn123456")
if (!is.null(file$file_handle)) {
  file_md5 = file$file_handle$content_md5

  # Find all entities with the same MD5
  matching_entities = synMd5Query(file_md5)
  print(sprintf("Found %d entities with same content:", length(matching_entities)))
  for (entity in matching_entities) {
    parent_id = if (!is.null(entity$parentId)) entity$parentId else "unknown"
    print(sprintf("- %s: %s in %s", entity$id, entity$name, parent_id))
  }
}

## End(Not run)

```

synMigrateIndexedFiles

Folder : migrate_indexed_files

Description

Migrate files that have been indexed with [synIndexFilesForMigration\(\)](#).

This is the second step in migrating files to a new storage location. Files must first be indexed using [synIndexFilesForMigration\(\)](#).

Interactive confirmation: When called from an interactive shell and `force=FALSE` (the default), this function will print the number of items queued for migration and prompt for confirmation before proceeding. If standard output is not connected to an interactive terminal (e.g. a script or CI environment), migration is aborted unless `force=TRUE` is set.

Usage

```
synMigrateIndexedFiles(instance, db_path, create_table_snapshots=TRUE, continue_on_error=FALSE, force)
```

Arguments

instance	(Folder) The Folder instance to operate on.
db_path	(character) Path to the SQLite database file created by synIndexFilesForMigration() . You can get this from the db_path field of the MigrationResult it returns.
create_table_snapshots	(logical) Whether to create table snapshots before migrating table files.
continue_on_error	(logical) Whether to continue migration if an error occurs.
force	(logical) Skip the interactive confirmation prompt and proceed with migration automatically. Set to TRUE when running non-interactively (scripts, CI, automated pipelines). Defaults to FALSE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A MigrationResult object containing migration statistics, or NULL if migration was aborted (user declined the confirmation prompt, or the session is non-interactive and force=FALSE).

Examples

```
## Not run:
## Migrating indexed files
# Migrate previously indexed files:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()

# Index first
index_result = folder |> synIndexFilesForMigration(
  dest_storage_location_id = 12345L
)

# Then migrate using the db_path from the index result
result = folder |> synMigrateIndexedFiles(
  db_path = index_result$db_path,
  force = TRUE # Skip interactive confirmation
)
print(result$counts_by_status)

## End(Not run)
```

 synMigrateIndexedFiles

Project : migrate_indexed_files

Description

Migrate files that have been indexed with `synIndexFilesForMigration()`.

This is the second step in migrating files to a new storage location. Files must first be indexed using `synIndexFilesForMigration()`.

Interactive confirmation: When called from an interactive shell and `force=FALSE` (the default), this function will print the number of items queued for migration and prompt for confirmation before proceeding. If standard output is not connected to an interactive terminal (e.g. a script or CI environment), migration is aborted unless `force=TRUE` is set.

Usage

```
synMigrateIndexedFiles(instance, db_path, create_table_snapshots=TRUE, continue_on_error=FALSE, force=FALSE)
```

Arguments

<code>instance</code>	(Project) The Project instance to operate on.
<code>db_path</code>	(character) Path to the SQLite database file created by <code>synIndexFilesForMigration()</code> . You can get this from the <code>db_path</code> field of the <code>MigrationResult</code> it returns.
<code>create_table_snapshots</code>	(logical) Whether to create table snapshots before migrating table files.
<code>continue_on_error</code>	(logical) Whether to continue migration if an error occurs.
<code>force</code>	(logical) Skip the interactive confirmation prompt and proceed with migration automatically. Set to <code>TRUE</code> when running non-interactively (scripts, CI, automated pipelines). Defaults to <code>FALSE</code> .
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A `MigrationResult` object containing migration statistics, or `NULL` if migration was aborted (user declined the confirmation prompt, or the session is non-interactive and `force=FALSE`).

Examples

```
## Not run:
## Migrating indexed files
# Migrate previously indexed files:
library(synapser)
synLogin()
```

```

project = Project(id = "syn123") |> synGet()

# Index first
index_result = project |> synIndexFilesForMigration(
  dest_storage_location_id = 12345L
)

# Then migrate using the db_path from the index result
result = project |> synMigrateIndexedFiles(
  db_path = index_result$db_path,
  force = TRUE # Skip interactive confirmation
)
print(result$counts_by_status)

## End(Not run)

```

synOnweb

*synOnweb***Description**

Open up a browser window to the entity page or wiki-subpage.

Usage

```
synOnweb(entity, subpage_id=NULL, synapse_client=NULL)
```

Arguments

entity	(Union[str, Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable]) Either an Entity object (Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable) or a Synapse ID string.
subpage_id	(Optional[str]) (Optional) ID of one of the wiki's sub-pages.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The URL that was opened in the browser.

Examples

```

## Not run:
## Example 1: Opening an entity in browser
# Open an entity's page in the default web browser:
library(synapser)
synLogin()

```

```

# Open by Synapse ID
url = synOnweb("syn123456")
print(sprintf("Opened: %s", url))

## Example 2: Opening with an entity object
# Open an entity using an entity object:
library(synapser)
synLogin()

file = synGet(synapse_id = "syn123456")
url = synOnweb(file)
print(sprintf("Opened file: %s", url))

## Example 3: Opening a wiki subpage
# Open a specific wiki subpage:
library(synapser)
synLogin()

# Open a specific wiki subpage
url = synOnweb("syn123456", subpage_id = "12345")
print(sprintf("Opened wiki subpage: %s", url))

## End(Not run)

```

synPrintEntity

synPrintEntity

Description

Pretty print an Entity as JSON.

Usage

```
synPrintEntity(entity, ensure_ascii=TRUE, synapse_client=NULL)
```

Arguments

entity	(Union[str, dict, Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable]) Either an Entity object (Dataset, DatasetCollection, EntityView, File, Folder, Link, MaterializedView, Project, RecordSet, SearchIndex, SubmissionView, Table, VirtualTable), a Synapse ID string, or a named list representation of an entity.
ensure_ascii	(bool) If TRUE, escapes all non-ASCII characters in the output. Defaults to TRUE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL. Prints the entity to the logger.

Examples

```
## Not run:
## Example 1: Printing an entity by ID
# Print an entity using its Synapse ID:
library(synapser)
synLogin()

# Print entity by Synapse ID
synPrintEntity("syn123456")

## Example 2: Printing an entity object
# Print an entity object:
library(synapser)
synLogin()

# Get and print a file entity
file = synGet(synapse_id = "syn123456")
synPrintEntity(file)

## Example 3: Controlling ASCII output
# Print an entity with non-ASCII characters:
library(synapser)
synLogin()

# Print with unicode characters preserved
synPrintEntity("syn123456", ensure_ascii = FALSE)

## End(Not run)
```

synPrompt

Agent : prompt

Description

Sends a prompt to the agent for the current session. If no session is currently active, a new session will be started.

Usage

```
synPrompt(instance, prompt, enable_trace=FALSE, print_response=FALSE, session=NULL, newer_than=NULL,
```

Arguments

instance	(Agent) The Agent instance to operate on.
prompt	(str) The prompt to send to the agent.
enable_trace	(bool) Whether to enable trace for the prompt.
print_response	(bool) Whether to print the response to the console.
session	(Optional[AgentSession]) The session to send the prompt to. If not provided, the current session will be used.
newer_than	(Optional[int]) The timestamp to get trace results newer than. Defaults to NULL (all results).
timeout	(int) The number of seconds to wait for the job to complete or progress before raising a SynapseTimeoutError. Defaults to 120.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Example 1: Prompt the baseline Synapse Agent.
# The baseline Synapse Agent is equivalent to the Agent available in the Synapse UI.
library(synapser)
synLogin()

my_agent = Agent()
my_agent |> synPrompt(
  prompt = "Can you tell me about the AD Knowledge Portal dataset?",
  enable_trace = TRUE,
  print_response = TRUE
)

## Example 2: Prompt a custom agent.
# If you have already registered a custom agent, you can prompt it by
# providing the agent's registration ID.
library(synapser)
synLogin()

my_agent = Agent(registration_id = "foo")
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## Advanced Example: Start and prompt multiple sessions
# Here, we connect to a custom agent and start one session with the prompt "Hello".
# In the background, this first session is being set as the current session
```

```

# and future prompts will be sent to this session by default. If we want to send a
# prompt to a different session, we can do so by starting it and calling synPrompt again,
# but with our new session as an argument. We now have two sessions, both stored in the
# my_agent$sessions named list. After the second prompt, my_second_session is now
# the current session.
library(synapser)
synLogin()

my_agent = Agent(registration_id = "foo")

my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

my_second_session = my_agent |> synStartSession()
my_agent |> synPrompt(
  prompt = "Hello again",
  enable_trace = TRUE,
  print_response = TRUE,
  session = my_second_session
)

## End(Not run)

```

synPrompt

AgentSession : prompt

Description

Sends a prompt to the agent and adds the response to the AgentSession's chat history. A session must be started before sending a prompt.

Usage

```
synPrompt(instance, prompt, enable_trace=FALSE, print_response=FALSE, newer_than=NULL, timeout=120L,
```

Arguments

instance	(AgentSession) The AgentSession instance to operate on.
prompt	(str) The prompt to send to the agent.
enable_trace	(bool) Whether to enable trace for the prompt.
print_response	(bool) Whether to print the response to the console.
newer_than	(Optional[int]) The timestamp to get trace results newer than. Defaults to NULL (all results).
timeout	(int) The number of seconds to wait for the job to complete or progress before raising a SynapseTimeoutError. Defaults to 120.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Send a prompt within an existing session.
# If you already have an existing session, you can send it a prompt by
# constructing an AgentSession with its session ID.
library(synapser)
synLogin()

my_session = AgentSession(id = "foo")
my_session |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## End(Not run)
```

synQuery	<i>Dataset : query</i>
----------	------------------------

Description

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a download_location in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

- query (character) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html>
- include_row_id_and_row_version (logical) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the data frame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.

convert_to_datetime	(logical) (data frame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects
download_location	(CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a data frame.
quote_character	(CSV Only) The character to use to quote fields. The default is a double quote.
escape_character	(CSV Only) The character to use to escape special characters. The default is a backslash.
line_end	(CSV Only) The character to use to end a line. The default is the system's line separator.
separator	(CSV Only) The character to use to separate fields. The default is a comma.
header	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
timeout	(integer) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if download_location is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a dataset and print out
# the results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
print(results)

## End(Not run)
```

synQuery

*DatasetCollection : query***Description**

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

query	(character) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
include_row_id_and_row_version	(logical) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the data frame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.
convert_to_datetime	(logical) (data frame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects
download_location	(CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a data frame.
quote_character	(CSV Only) The character to use to quote fields. The default is a double quote.
escape_character	(CSV Only) The character to use to escape special characters. The default is a backslash.
line_end	(CSV Only) The character to use to end a line. The default is the system's line separator.
separator	(CSV Only) The character to use to separate fields. The default is a comma.
header	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
timeout	(integer) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if download_location is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a dataset collection and
# print out the results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
print(results)

## End(Not run)
```

synQuery	<i>EntityView : query</i>
----------	---------------------------

Description

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a download_location in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

- query (str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html>
- include_row_id_and_row_version (bool) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the data frame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.
- convert_to_datetime (bool) (data frame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects

download_location	(CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a data frame.
quote_character	(CSV Only) The character to use to quote fields. The default is a double quote.
escape_character	(CSV Only) The character to use to escape special characters. The default is a backslash.
line_end	(CSV Only) The character to use to end a line. The default is the system's line separator.
separator	(CSV Only) The character to use to separate fields. The default is a comma.
header	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
timeout	(int) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if download_location is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in an EntityView and print out the
# results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
results

## End(Not run)
```

synQuery

MaterializedView : query

Description

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a download_location in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a DataFrame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

query (str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html>

include_row_id_and_row_version (bool) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the DataFrame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.

convert_to_datetime (bool) (DataFrame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects

download_location (CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a DataFrame.

quote_character (CSV Only) The character to use to quote fields. The default is a double quote.

escape_character (CSV Only) The character to use to escape special characters. The default is a backslash.

line_end (CSV Only) The character to use to end a line. The default is the system's line separator.

separator (CSV Only) The character to use to separate fields. The default is a comma.

header (CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.

timeout (int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if `download_location` is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a materialized view and print
# out the results.
library(synapser)
```

```
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
results

## End(Not run)
```

synQuery	<i>SubmissionView : query</i>
----------	-------------------------------

Description

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a download_location in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a DataFrame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

- query (str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html>>
- include_row_id_and_row_version (bool) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the DataFrame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.
- convert_to_datetime (bool) (DataFrame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects
- download_location (CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a DataFrame.
- quote_character (CSV Only) The character to use to quote fields. The default is a double quote.
- escape_character (CSV Only) The character to use to escape special characters. The default is a backslash.
- line_end (CSV Only) The character to use to end a line. The default is the system's line separator.
- separator (CSV Only) The character to use to separate fields. The default is a comma.

header	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
timeout	(int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if `download_location` is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a submission view and print
# out the results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
results

## End(Not run)
```

synQuery

Table : query

Description

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
-------	--

<code>include_row_id_and_row_version</code>	(bool) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the data frame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.
<code>convert_to_datetime</code>	(bool) (data frame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects
<code>download_location</code>	(CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a data frame.
<code>quote_character</code>	(CSV Only) The character to use to quote fields. The default is a double quote.
<code>escape_character</code>	(CSV Only) The character to use to escape special characters. The default is a backslash.
<code>line_end</code>	(CSV Only) The character to use to end a line. The default is the system's line separator.
<code>separator</code>	(CSV Only) The character to use to separate fields. The default is a comma.
<code>header</code>	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
<code>timeout</code>	(int) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if `download_location` is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a table and print out the
# results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
results

## End(Not run)
```


synQuery

*VirtualTable : query***Description**

Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a `DataFrame` or downloading the results to a file.

Usage

```
synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NU
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
include_row_id_and_row_version	(bool) If TRUE the ROW_ID and ROW_VERSION columns will be returned in the DataFrame. These columns are required if using the query results to update rows in the table. These columns are the primary keys used by Synapse to uniquely identify rows in the table.
convert_to_datetime	(bool) (DataFrame only) If set to TRUE, will convert all Synapse DATE columns from UNIX timestamp integers into UTC datetime objects
download_location	(CSV Only) If set to a path the results will be downloaded to that directory. The results will be downloaded as a CSV file. A path to the downloaded file will be returned instead of a DataFrame.
quote_character	(CSV Only) The character to use to quote fields. The default is a double quote.
escape_character	(CSV Only) The character to use to escape special characters. The default is a backslash.
line_end	(CSV Only) The character to use to end a line. The default is the system's line separator.
separator	(CSV Only) The character to use to separate fields. The default is a comma.
header	(CSV Only) If set to TRUE the first row will be used as the header row. The default is TRUE.
timeout	(int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a data frame or a path to the downloaded query results if download_location is set.

Examples

```
## Not run:
## Querying for data
# This example shows how you may query for data in a table and print out the
# results.
library(synapser)
synLogin()

results = synQuery(query = "SELECT * FROM syn1234")
results

## End(Not run)
```

synQueryPartMask	<i>Dataset : query_part_mask</i>
------------------	----------------------------------

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the [synQuery\(\)](#) function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

- | | |
|-----------|--|
| query | (character) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html |
| part_mask | (integer) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none"> - Query Results (queryResults) = 0x1 - Query Count (queryCount) = 0x2 - The sum of the file sizes (sumFileSizesBytes) = 0x40 - The last updated on date of the table (lastUpdatedOn) = 0x80 |

timeout (integer) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a dataset
# and print out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synpser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
print(result)

## End(Not run)
```

synQueryPartMask

DatasetCollection : query_part_mask

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the [synQuery\(\)](#) function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(character) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
part_mask	(integer) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none"> - Query Results (queryResults) = 0x1 - Query Count (queryCount) = 0x2 - The sum of the file sizes (sumFileSizesBytes) = 0x40 - The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(integer) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a dataset collection
# and print out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
print(result)

## End(Not run)
```

synQueryPartMask	<i>EntityView : query_part_mask</i>
------------------	-------------------------------------

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
part_mask	(int) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none"> - Query Results (queryResults) = 0x1 - Query Count (queryCount) = 0x2 - The sum of the file sizes (sumFileSizesBytes) = 0x40 - The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(int) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in an EntityView and print
# out the results.

# In this case we are getting the results of the query, the count of rows, and
```

```
# the last updated on date of the EntityView.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
result

## End(Not run)
```

synQueryPartMask	<i>MaterializedView : query_part_mask</i>
------------------	---

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the [synQuery\(\)](#) function.

The query for this method uses this Rest API: <<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: < https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html >
part_mask	(int) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none">- Query Results (queryResults) = 0x1- Query Count (queryCount) = 0x2- The sum of the file sizes (sumFileSizesBytes) = 0x40- The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a materialized view
# and print out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
result

## End(Not run)
```

synQueryPartMask

SubmissionView : query_part_mask

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the [synQuery\(\)](#) function.

The query for this method uses this Rest API: <<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: < https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html >
-------	--

part_mask	(int) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none">- Query Results (queryResults) = 0x1- Query Count (queryCount) = 0x2- The sum of the file sizes (sumFileSizesBytes) = 0x40- The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a submission view
# and print out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
result

## End(Not run)
```

synQueryPartMask	Table : query_part_mask
------------------	-------------------------

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should

also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API: <https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html
part_mask	(int) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: <ul style="list-style-type: none"> - Query Results (queryResults) = 0x1 - Query Count (queryCount) = 0x2 - The sum of the file sizes (sumFileSizesBytes) = 0x40 - The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(int) The number of seconds to wait for the query job to complete or progress before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a table and print
# out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)
```

```
result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
result

## End(Not run)
```

synQueryPartMask	<i>VirtualTable : query_part_mask</i>
------------------	---------------------------------------

Description

Query for data on a table stored in Synapse. This is a more advanced use case of the [synQuery\(\)](#) function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the [synQuery\(\)](#) function.

The query for this method uses this Rest API: <<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

Usage

```
synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)
```

Arguments

query	(str) The query to run. The query must be valid syntax that Synapse can understand. See this document that describes the expected syntax of the query: < https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/web/controller/TableExamples.html >
part_mask	(int) The bitwise OR of the part mask values you want to return in the results. The following list of part masks are implemented to be returned in the results: - Query Results (queryResults) = 0x1 - Query Count (queryCount) = 0x2 - The sum of the file sizes (sumFileSizesBytes) = 0x40 - The last updated on date of the table (lastUpdatedOn) = 0x80
timeout	(int) The number of seconds to wait for the query job to complete before raising a timeout error. Defaults to 250.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The results of the query as a QueryResultOutput object.

Examples

```
## Not run:
## Querying for data with a part mask
# This example shows how to use the bitwise OR to combine the
# part mask values and then use that to query for data in a table and print
# out the results.

# In this case we are getting the results of the query, the count of rows, and
# the last updated on date of the table.
library(synapser)
synLogin()

QUERY_RESULTS = 0x1
QUERY_COUNT = 0x2
LAST_UPDATED_ON = 0x80

# Combine the part mask values using bitwise OR
part_mask = bitwOr(bitwOr(QUERY_RESULTS, QUERY_COUNT), LAST_UPDATED_ON)

result = synQueryPartMask(query = "SELECT * FROM syn1234", part_mask = part_mask)
result

## End(Not run)
```

synReformatAttachmentFileName

WikiPage : reformat_attachment_file_name

Description

Reformat the attachment file name to be a valid attachment path.

Usage

```
synReformatAttachmentFileName(attachment_file_name)
```

Arguments

attachment_file_name
(str) The name of the attachment file.

Value

The reformatted attachment file name.

Examples

```
## Not run:
## Reformat the attachment file name
attachment_file_name = "file.txt"
attachment_file_name_reformatted = synReformatAttachmentFileName(attachment_file_name)
cat("Reformatted attachment file name:", attachment_file_name_reformatted, "\n")

## End(Not run)
```

synRegister

Agent : register

Description

Registers an agent with the Synapse API. If agent already exists, it will be retrieved.

Usage

```
synRegister(instance, synapse_client=NULL)
```

Arguments

`instance` (Agent) The Agent instance to operate on.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The registered or existing Agent object.

Examples

```
## Not run:
## Register and chat with a custom agent
# Only available for internal users (Sage Bionetworks employees).
#
# Alternatively, you can register a custom agent and chat with it provided
# you have already created it.
library(synapser)
synLogin()

my_agent = Agent(cloud_agent_id = "foo") |> synRegister()
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## End(Not run)
```

synRemoveItem	<i>Dataset : remove_item</i>
---------------	------------------------------

Description

Removes an item from the dataset. For Folders, all children of the folder are removed recursively. Effect is not seen until the dataset is stored.

Usage

```
synRemoveItem(instance, item, synapse_client=NULL)
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
item	(EntityRef, File, or Folder) The entity to remove from the dataset. Must be an EntityRef , File, or Folder.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Errors

Raises an error if the item is not an EntityRef, File, or Folder.

Examples

```
## Not run:
## Example 1: Remove a file from a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synRemoveItem(File(id = "syn1235"))
my_dataset |> synStore()

## Example 2: Remove a folder from a dataset.
# All child files are recursively removed from the dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synRemoveItem(Folder(id = "syn1236"))
my_dataset |> synStore()

## Example 3: Remove an entity reference from a dataset.
```

```

library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synRemoveItem(EntityRef(id = "syn1237", version = 1L))
my_dataset |> synStore()

## End(Not run)

```

synRemoveItem	<i>DatasetCollection : remove_item</i>
---------------	--

Description

Removes an entity from the dataset collection. Must be a Dataset or EntityRef. Effect is not seen until the dataset collection is stored. Unless the version is specified, all entities with the same ID will be removed.

Usage

```
synRemoveItem(instance, item)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
item	(Dataset or EntityRef) The Dataset, or an EntityRef pointing to one, to remove from the collection.

Value

NULL

Errors

Raises an error if the item is not a Dataset or EntityRef.

Examples

```

## Not run:
## Example 1: Remove a Dataset from a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synRemoveItem(Dataset(id = "syn1235", version_number = 1L))
my_collection |> synStore()

## Example 2: Remove all versions of a Dataset from a Dataset Collection.
library(synapser)

```

```

synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synRemoveItem(Dataset(id = "syn1235"))
my_collection |> synStore()

## End(Not run)

```

synReorderColumn	<i>Dataset : reorder_column</i>
------------------	---------------------------------

Description

Reorder a column in the table. Note that this does not store the column in Synapse. You must call the [synStore\(\)](#) function on this Dataset instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

Usage

```
synReorderColumn(instance, name, index)
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
name	(character) The name of the column to reorder.
index	(integer) The index to move the column to starting with 0.

Value

NULL

Examples

```

## Not run:
## Reordering a column
# This example shows how you may reorder a column in a dataset and then
# store the change back in Synapse.
library(synapser)
synLogin()

dataset = Dataset(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

# Move the column to the beginning of the list
dataset |> synReorderColumn(name = "my_column", index = 0L)
dataset |> synStore()

## End(Not run)

```

synReorderColumn	<i>DatasetCollection : reorder_column</i>
------------------	---

Description

Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this `DatasetCollection` instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

Usage

```
synReorderColumn(instance, name, index)
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
name	(character) The name of the column to reorder.
index	(integer) The index to move the column to starting with 0.

Value

NULL

Examples

```
## Not run:
## Reordering a column
# This example shows how you may reorder a column in a dataset collection and
# then store the change back in Synapse.
library(synapser)
synLogin()

collection = DatasetCollection(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

# Move the column to the beginning of the list
collection |> synReorderColumn(name = "my_column", index = 0L)
collection |> synStore()

## End(Not run)
```

synReorderColumn	<i>EntityView : reorder_column</i>
------------------	------------------------------------

Description

Reorder a column in the EntityView. Note that this does not store the column in Synapse. You must call the [synStore\(\)](#) function on this EntityView instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the \$columns attribute.

You must ensure that the index is within the bounds of the number of columns in the EntityView. If you pass in an index that is out of bounds the column will be added to the end of the list.

Usage

```
synReorderColumn(instance, name, index)
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
name	(str) The name of the column to reorder.
index	(int) The index to move the column to starting with 0.

Value

NULL

Examples

```
## Not run:
## Reordering a column
# This example shows how you may reorder a column in an EntityView and then store
# the change back in Synapse.

library(synapser)
synLogin()

entityview = EntityView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

# Move the column to the beginning of the list
entityview |> synReorderColumn(name = "my_column", index = 0L)
entityview |> synStore()

## End(Not run)
```

synReorderColumn	<i>SubmissionView : reorder_column</i>
------------------	--

Description

Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this SubmissionView instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

Usage

```
synReorderColumn(instance, name, index)
```

Arguments

instance	(SubmissionView) The SubmissionView instance to operate on.
name	(str) The name of the column to reorder.
index	(int) The index to move the column to starting with 0.

Value

NULL

Examples

```
## Not run:
## Reordering a column
# This example shows how you may reorder a column in a submission view and
# then store the change back in Synapse.
library(synapser)
synLogin()

view = SubmissionView(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

# Move the column to the beginning of the list
view |> synReorderColumn(name = "my_column", index = 0L)
view |> synStore()

## End(Not run)
```

synReorderColumn	Table : reorder_column
------------------	------------------------

Description

Reorder a column in the table. Note that this does not store the column in Synapse. You must call the [synStore\(\)](#) function on this Table instance to store the column in Synapse. This is a convenience function to eliminate the need to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

Usage

```
synReorderColumn(instance, name, index)
```

Arguments

instance	(Table) The Table instance to operate on.
name	(str) The name of the column to reorder.
index	(int) The index to move the column to starting with 0.

Value

NULL

Examples

```
## Not run:
## Reordering a column
# This example shows how you may reorder a column in a table and then store
# the change back in Synapse.
library(synapser)
synLogin()

table = Table(id = "syn1234") |> synGet(table_options = TableOptions(include_columns = TRUE))

# Move the column to the beginning of the list
table |> synReorderColumn(name = "my_column", index = 0L)
table |> synStore()

## End(Not run)
```

synRestDelete	<i>synRestDelete</i>
---------------	----------------------

Description

Sends an HTTP DELETE request to the Synapse server.

Usage

`synRestDelete(uri, endpoint=NULL, headers=NULL, retry_policy=list(), requests_session_async_synapse=)`

Arguments

- `uri` (str) URI of resource to be deleted
- `endpoint` (str) Server endpoint, defaults to the client’s configured repository endpoint (see [synSetEndpoints\(\)](#)).
- `headers` (Headers) Named list of headers to use.
- `retry_policy` (dict[str, Any]) A retry policy (named list) whose keys should match the parameters of `with_retry_time_based_async`, the underlying Python client’s retry mechanism.
- `requests_session_async_synapse` (AsyncClient) The async client to use when making this specific call
- `kwargs` optional named parameter: Any other arguments taken by a `request` method

Value

NULL

Note

This wraps `synapseclient’s rest_delete_async()`, which is a Python `async` coroutine method. Calling `synRestDelete()` does not run that coroutine – it returns an unresolved Python coroutine object, since `reticulate` does not automatically await coroutines. To get the actual result, resolve the returned coroutine yourself with Python’s `asyncio.run()` via `reticulate`, as shown in the example below.

Examples

```
## Not run:
## Using this function
# For ordinary entity deletion, prefer the higher-level synDelete()
# function instead -- e.g. synDelete("syn123456"). Reach for
# synRestDelete() only for REST endpoints that don't have a
# higher-level model wrapper.
library(synapser)
synLogin()
```

```

    asyncio = reticulate::import("asyncio")

    coro = synRestDelete(uri = "/entity/syn123456")
    response = asyncio$run(coro)
    response

## End(Not run)

```

synRestGet

synRestGet

Description

Sends an HTTP GET request to the Synapse server.

Usage

```
synRestGet(uri, endpoint=NULL, headers=NULL, retry_policy=list(), requests_session_async_synapse=NULL)
```

Arguments

uri	(str) URI on which get is performed
endpoint	(str) Server endpoint, defaults to the client's configured repository endpoint (see synSetEndpoints()).
headers	(Headers) Named list of headers to use.
retry_policy	(dict[str, Any]) A retry policy (named list) whose keys should match the parameters of with_retry_time_based_async , the underlying Python client's retry mechanism.
requests_session_async_synapse	(AsyncClient) The async client to use when making this specific call.
kwargs	optional named parameter: Any other arguments taken by a request method

Value

JSON encoding of response

Note

This wraps synapseclient's `rest_get_async()`, which is a Python async coroutine method. Calling `synRestGet()` does not run that coroutine – it returns an unresolved Python coroutine object, since `reticulate` does not automatically await coroutines. To get the actual response, resolve the returned coroutine yourself with Python's `asyncio.run()` via `reticulate`, as shown in the example below.

Examples

```
## Not run:
## Using this function
library(synapser)
synLogin()

asyncio = reticulate::import("asyncio")

# Low-level REST access to an entity's raw JSON, bypassing the synGet() model wrapper
coro = synRestGet(uri = "/entity/syn123456")
response = asyncio$run(coro)
response

## End(Not run)
```

synRestore

WikiPage : restore

Description

Restore a specific version of a wiki page.

Usage

```
synRestore(instance, synapse_client=NULL)
```

Arguments

instance (WikiPage) The WikiPage instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The restored wiki page.

Examples

```
## Not run:
## Restore a specific version of a wiki page
# This example shows how to restore a specific version of a wiki page.
wiki_page_restored = WikiPage(owner_id = project$id, id = wiki_page$id, wiki_version = "0") |> synRestore()
wiki_page_restored

## End(Not run)
```

synRestPost	<i>synRestPost</i>
-------------	--------------------

Description

Sends an HTTP POST request to the Synapse server.

Usage

```
synRestPost(uri, body=NULL, endpoint=NULL, headers=NULL, retry_policy=list(), requests_session_async=)
```

Arguments

uri	(str) URI on which the request is performed
body	(Any) The payload to be delivered, as a JSON-encoded string (e.g. via <code>rjson::toJSON()</code>). This is passed straight through to the underlying HTTP client's content argument, which only accepts a string or raw bytes – passing a named list/dict directly will fail with <code>TypeError: Unexpected type for 'content'</code> .
endpoint	(str) Server endpoint, defaults to the client's configured repository endpoint (see synSetEndpoints()).
headers	(Headers) Named list of headers to use.
retry_policy	(dict[str, Any]) A retry policy (named list) whose keys should match the parameters of <code>with_retry_time_based_async</code> , the underlying Python client's retry mechanism.
requests_session_async_synapse	(AsyncClient) The async client to use when making this specific call.
kwargs	optional named parameter: Any other arguments taken by a <code>request</code> method

Value

JSON encoding of response

Note

This wraps `synapseclient's rest_post_async()`, which is a Python async coroutine method. Calling `synRestPost()` does not run that coroutine – it returns an unresolved Python coroutine object, since `reticulate` does not automatically await coroutines. To get the actual response, resolve the returned coroutine yourself with Python's `asyncio.run()` via `reticulate`, as shown in the example below.

Examples

```
## Not run:
## Using this function
# For ordinary entity creation, prefer the higher-level constructors
# and synStore() instead -- e.g. Folder(...) |> synStore(). Reach for
```

```
# synRestPost() only for REST endpoints that don't have a higher-level
# model wrapper.
library(synapser)
synLogin()

asyncio = reticulate::import("asyncio")

new_folder_properties = list(
  name = "My New Folder",
  parentId = "syn123456",
  concreteType = "org.sagebionetworks.repo.model.Folder"
)
post_coro = synRestPost(uri = "/entity", body = rjson::toJSON(new_folder_properties))
response = asyncio$run(post_coro)
response

## End(Not run)
```

synRestPut	<i>synRestPut</i>
------------	-------------------

Description

Sends an HTTP PUT request to the Synapse server.

Usage

```
synRestPut(uri, body=NULL, endpoint=NULL, headers=NULL, retry_policy=list(), requests_session_async_s
```

Arguments

uri	(str) URI on which the request is performed
body	(Any) The payload to be delivered, as a JSON-encoded string (e.g. via <code>rjson::toJSON()</code>). This is passed straight through to the underlying HTTP client's content argument, which only accepts a string or raw bytes – passing a named list/dict directly will fail with <code>TypeError: Unexpected type for 'content'</code> .
endpoint	(str) Server endpoint, defaults to the client's configured repository endpoint (see synSetEndpoints()).
headers	(Headers) Named list of headers to use.
retry_policy	(dict[str, Any]) A retry policy (named list) whose keys should match the parameters of <code>with_retry_time_based_async</code> , the underlying Python client's retry mechanism.
requests_session_async_synapse	(AsyncClient) The async client to use when making this specific call.
kwargs	optional named parameter: Any other arguments taken by a <code>request</code> method

Value

JSON encoding of response

Note

This wraps synapseclient's `rest_put_async()`, which is a Python async coroutine method. Calling `synRestPut()` does not run that coroutine – it returns an unresolved Python coroutine object, since `reticulate` does not automatically await coroutines. To get the actual response, resolve the returned coroutine yourself with Python's `asyncio.run()` via `reticulate`, as shown in the example below.

Examples

```
## Not run:
## Using this function
# For ordinary entity updates, prefer the higher-level synSetProperties()
# / synStore() functions instead -- they handle this fetch-modify-store
# cycle for you. Reach for synRestPut() only for REST endpoints that
# don't have a higher-level model wrapper.
library(synapser)
synLogin()

asyncio = reticulate::import("asyncio")

# Fetch the entity's current properties first
get_coro = synRestGet(uri = "/entity/syn123456")
entity_properties = asyncio$run(get_coro)

# Modify the field(s) you want to change
entity_properties$name = "New Name"

# The body must be a JSON-encoded string, not a named list -- it is passed
# straight through to the underlying HTTP client's `content` argument,
# which only accepts a string or raw bytes.
put_coro = synRestPut(uri = "/entity/syn123456", body = rjson::toJSON(entity_properties))
response = asyncio$run(put_coro)

## End(Not run)
```

synSendMessage

synSendMessage

Description

Send a message via Synapse.

Usage

```
synSendMessage(userIds, messageSubject, messageBody, contentType="text/plain")
```

Arguments

- userIds A list of user IDs to which the message is to be sent
- messageSubject The subject for the message
- messageBody The body of the message
- contentType optional contentType of message body (default="text/plain") Should be one of "text/plain" or "text/html"

Value

The metadata of the created message

Examples

```
## Not run:
## Sending a message
library(synapser)
synLogin()

message = synSendMessage(
  userIds = list(273948L),
  messageSubject = "Data is ready",
  messageBody = "The analysis results have been uploaded to syn123."
)

# Send an HTML message
message = synSendMessage(
  userIds = list(273948L),
  messageSubject = "Data is ready",
  messageBody = "<p>The results are in <b>syn123</b>.</p>",
  contentType = "text/html"
)

## End(Not run)
```

synSetEndpoints	<i>synSetEndpoints</i>
-----------------	------------------------

Description

Sets the locations for each of the Synapse services (mostly useful for testing).

Usage

```
synSetEndpoints(repoEndpoint=NULL, authEndpoint=NULL, fileHandleEndpoint=NULL, portalEndpoint=NULL, s
```

Arguments

repoEndpoint (str) Location of synapse repository
 authEndpoint (str) Location of authentication service
 fileHandleEndpoint (str) Location of file service
 portalEndpoint (str) Location of the website
 skip_checks (bool) Skip version and endpoint checks

Value

NULL

Examples

```

## Not run:
## Switching endpoints
# To switch between staging and production endpoints
library(synapser)

synSetEndpoints(
  repoEndpoint = "https://repo-staging.prod.sagebase.org/repo/v1",
  authEndpoint = "https://auth-staging.prod.sagebase.org/auth/v1",
  fileHandleEndpoint = "https://file-staging.prod.sagebase.org/file/v1",
  portalEndpoint = "https://staging.synapse.org/"
)

synSetEndpoints(
  repoEndpoint = "https://repo-prod.prod.sagebase.org/repo/v1",
  authEndpoint = "https://auth-prod.prod.sagebase.org/auth/v1",
  fileHandleEndpoint = "https://file-prod.prod.sagebase.org/file/v1",
  portalEndpoint = "https://www.synapse.org/"
)

## End(Not run)

```

synSetPermissions	<i>Dataset : set_permissions</i>
-------------------	----------------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance	(Dataset) The Dataset instance to operate on.
principal_id	(integer) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[character]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(logical) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(logical) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(logical) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList.html>.

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

Dataset(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
Dataset(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>DatasetCollection : set_permissions</i>
-------------------	--

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance	(DatasetCollection) The DatasetCollection instance to operate on.
principal_id	(integer) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[character]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(logical) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(logical) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(logical) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControllist.html>.

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

DatasetCollection(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNL

# Grant the public view access
DatasetCollection(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>EntityView : set_permissions</i>
-------------------	-------------------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

- instance (EntityView) The EntityView instance to operate on.
- principal_id (int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
- access_type (list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

- modify_benefactor (bool) Set as TRUE when modifying a benefactor’s ACL. The term ’benefactor’ is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity’s benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
- warn_if_inherits (bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it’s the parent folder, or another entity in the parent tree.

overwrite (bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList.html>.

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

EntityView(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
EntityView(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions *File : set_permissions*

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance (File) The File instance to operate on.

principal_id (int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.

access_type (list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An **Access Control List** object.

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

File(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
File(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>Folder : set_permissions</i>
-------------------	---------------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```


Arguments

instance	(Folder) The Folder instance to operate on.
principal_id	(int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An **Access Control List** object.

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

Folder(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
Folder(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>MaterializedView : set_permissions</i>
-------------------	---

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance	(MaterializedView) The MaterializedView instance to operate on.
principal_id	(int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList>>

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

MaterializedView(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLO

# Grant the public view access
MaterializedView(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>Project : set_permissions</i>
-------------------	----------------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance	(Project) The Project instance to operate on.
principal_id	(int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
-------------------	---

warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
------------------	---

- overwrite (bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList>>

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

Project(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
Project(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>SubmissionView : set_permissions</i>
-------------------	---

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh

Arguments

- instance (SubmissionView) The SubmissionView instance to operate on.
 - principal_id (int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
 - access_type (list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.
- Defaults to list("READ", "DOWNLOAD")**

- modify_benefactor** (bool) Set as TRUE when modifying a benefactor’s ACL. The term ‘benefactor’ is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity’s benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
- warn_if_inherits** (bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it’s the parent folder, or another entity in the parent tree.
- overwrite** (bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
- synapse_client** (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList>>

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

SubmissionView(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
SubmissionView(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	Table : set_permissions
-------------------	-------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=FALSE)

Arguments

instance	(Table) The Table instance to operate on.
principal_id	(int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList>>

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

Table(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
Table(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetPermissions	<i>VirtualTable : set_permissions</i>
-------------------	---------------------------------------

Description

Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Usage

```
synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inh
```

Arguments

instance	(VirtualTable) The VirtualTable instance to operate on.
principal_id	(int) Identifier of a user or group. 273948 is for all registered Synapse users and 273949 is for public access. NULL implies public access.
access_type	(list[str]) Type of permission to be granted. One or more of CREATE, READ, DOWNLOAD, UPDATE, DELETE, CHANGE_PERMISSIONS.

Defaults to list("READ", "DOWNLOAD")

modify_benefactor	(bool) Set as TRUE when modifying a benefactor's ACL. The term 'benefactor' is used to indicate which Entity an Entity inherits its ACL from. For example, a newly created Project will be its own benefactor, while a new FileEntity's benefactor will start off as its containing Project. If the entity already has local sharing settings the benefactor would be itself. It may also be the immediate parent, somewhere in the parent tree, or the project itself.
warn_if_inherits	(bool) When modify_benefactor is TRUE, this does not have any effect. When modify_benefactor is FALSE, and warn_if_inherits is TRUE, a warning log message is produced if the benefactor for the entity you passed into the function is not itself, i.e., it's the parent folder, or another entity in the parent tree.
overwrite	(bool) By default this function overwrites existing permissions for the specified user. Set this flag to FALSE to add new permissions non-destructively.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

An Access Control List object matching <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/AccessControlList>

Examples

```
## Not run:
## Setting permissions
# Grant all registered users download access
library(synapser)
synLogin()

VirtualTable(id = "syn123") |> synSetPermissions(principal_id = 273948L, access_type = list("READ", "DOWNLOAD"))

# Grant the public view access
VirtualTable(id = "syn123") |> synSetPermissions(principal_id = 273949L, access_type = list("READ"))

## End(Not run)
```

synSetProperties	<i>Set properties of an entity</i>
------------------	------------------------------------

Description

Set one or more properties of a Synapse entity object, such as its name, description, version_label, activity, or annotations, and return the entity so it can be piped into [synStore\(\)](#).

The changes are only made to the local entity object; call [synStore\(\)](#) to save them to Synapse.

Each argument must be named, and every name must be an attribute of the entity's model class (for example a field of [File](#) or [Folder](#)); otherwise an error is raised listing the names that are not valid. When duplicate properties are provided, the first one will be used. Calling [synSetProperties\(\)](#) with no properties returns the entity unchanged.

Setting annotations replaces all of the entity's annotations. To add to the existing annotations, combine them first, e.g. `annotations = append(entity$annotations, list(key = "value"))`; to remove all annotations, set `annotations = list()`.

Usage

```
synSetProperties(entity, ...)
```

Arguments

entity	A Synapse entity object, for example one created with File() , Folder() or Project() , or returned by synGet() .
...	Named properties to set, e.g. <code>name = "new name"</code> or <code>annotations = list(key = "value")</code> .

Value

The entity with the properties set.

Examples

```
## Not run:
synSetProperties(entity, annotations = list(key = "value"))
synSetProperties(entity, name = "new name", annotations = list(key = "value", key2 = "value2"))

## End(Not run)
```

synSetStorageLocation *Folder : set_storage_location*

Description

Set the upload storage location for this entity. This configures where files uploaded to this entity will be stored.

This is a destructive update. The provided storage_location_id value(s) will **replace** any storage locations previously configured on this entity. To add a storage location without removing existing ones, first retrieve the current setting with [synGetProjectSetting\(\)](#), then pass its existing locations together with the new ID as storage_location_id. The first ID in the list is the default upload destination.

Usage

```
synSetStorageLocation(instance, storage_location_id=1L, synapse_client=NULL)
```

Arguments

instance	(Folder) The Folder instance to operate on.
storage_location_id	(Optional[integer or list of integer]) The storage location ID(s) to set. Can be a single ID, a list of IDs (first is default, max 10). By default, the default Synapse S3 storage location is used.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The ProjectSetting object reflecting the current state after the operation.

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Example 1: Replace all storage locations
# Fully replace the storage location on a folder with a single location:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
setting = folder |> synSetStorageLocation(storage_location_id = 12345L)
print(setting)

## Example 2: Partial update – add a storage location without removing existing ones
# Retrieve the current setting and set the existing locations plus a new one:
library(synapser)
synLogin()

folder = Folder(id = "syn123") |> synGet()
setting = folder |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  locations = c(unlist(setting$locations), 67890L)
  folder |> synSetStorageLocation(storage_location_id = as.list(locations))
}

## End(Not run)
```

synSetStorageLocation *Project: set_storage_location*

Description

Set the upload storage location for this entity. This configures where files uploaded to this entity will be stored.

This is a destructive update. The provided `storage_location_id` value(s) will **replace** any storage locations previously configured on this entity. To add a storage location without removing existing ones, first retrieve the current setting with [synGetProjectSetting\(\)](#), then pass its existing locations together with the new ID as `storage_location_id`. The first ID in the list is the default upload destination.

Usage

```
synSetStorageLocation(instance, storage_location_id=1L, synapse_client=NULL)
```

Arguments

`instance` (Project) The Project instance to operate on.

`storage_location_id` (Optional[integer or list of integer]) The storage location ID(s) to set. Can be a single ID, a list of IDs (first is default, max 10). By default, the default Synapse S3 storage location is used.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The ProjectSetting object reflecting the current state after the operation.

Errors

Raises an error if the entity does not have an id set.

Examples

```
## Not run:
## Example 1: Replace all storage locations
# Fully replace the storage location on a project with a single location:
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
setting = project |> synSetStorageLocation(storage_location_id = 12345L)
print(setting)

## Example 2: Partial update – add a storage location without removing existing ones
# Retrieve the current setting and set the existing locations plus a new one:
library(synapser)
synLogin()

project = Project(id = "syn123") |> synGet()
setting = project |> synGetProjectSetting(setting_type = "upload")
if (!is.null(setting)) {
  locations = c(unlist(setting$locations), 67890L)
  project |> synSetStorageLocation(storage_location_id = as.list(locations))
}

## End(Not run)
```

synSnapshot	<i>Dataset : snapshot</i>
-------------	---------------------------

Description

Creates a snapshot of the dataset. A snapshot is a saved, read-only version of the dataset at the time it was created. Dataset snapshots are created using the asynchronous job API.

Usage

synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_versions=TRUE)

Arguments

instance	(Dataset) The Dataset instance to operate on.
comment	(Optional[character]) A unique comment to associate with the snapshot.
label	(Optional[character]) A unique label to associate with the snapshot.
include_activity	(logical) If TRUE the activity will be included in snapshot if it exists. In order to include the activity, the activity must have already been stored in Synapse by using the \$activity attribute on the Dataset and calling the synStore() function on the Dataset instance. Adding an activity to a snapshot of a dataset is meant to capture the provenance of the data at the time of the snapshot. Defaults to TRUE.
associate_activity_to_new_version	(logical) If TRUE the activity will be associated with the new version of the dataset. If FALSE the activity will not be associated with the new version of the dataset. Defaults to TRUE.
timeout	(integer) The number of seconds to wait for the async job to complete. Defaults to 120.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A TableUpdateTransaction object which includes the version number of the snapshot.

Examples

```
## Not run:
## Save a snapshot of a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()
my_dataset |> synSnapshot(comment = "My first snapshot", label = "My first snapshot")

## End(Not run)
```

synSnapshot

DatasetCollection : snapshot

Description

Creates a snapshot of the dataset collection. A snapshot is a saved, read-only version of the dataset collection at the time it was created. Dataset collection snapshots are created using the asynchronous job API.

Usage

```
synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_version=TRUE)
```

Arguments

instance (DatasetCollection) The DatasetCollection instance to operate on.

comment (Optional[character]) A unique comment to associate with the snapshot.

label (Optional[character]) A unique label to associate with the snapshot.

include_activity (logical) If TRUE the activity will be included in snapshot if it exists. In order to include the activity, the activity must have already been stored in Synapse by using the `$activity` attribute on the Dataset Collection and calling the `synStore()` function on the Dataset Collection instance. Adding an activity to a snapshot of a dataset collection is meant to capture the provenance of the data at the time of the snapshot. Defaults to TRUE.

associate_activity_to_new_version (logical) If TRUE the activity will be associated with the new version of the dataset collection. If FALSE the activity will not be associated with the new version of the dataset collection. Defaults to TRUE.

timeout (integer) The number of seconds to wait for the async job to complete. Defaults to 120.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A TableUpdateTransaction object which includes the version number of the snapshot.

Examples

```
## Not run:
## Save a snapshot of a Dataset Collection.
library(synapser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()
my_collection |> synSnapshot(comment = "My first snapshot", label = "My first snapshot")

## End(Not run)
```

synSnapshot	<i>EntityView : snapshot</i>
-------------	------------------------------

Description

Creates a snapshot of the EntityView entity. Synapse handles snapshot creation differently for Table- and View-like entities. View snapshots are created using the asynchronous job API.

Making a snapshot of a view allows you to create an immutable version of the view at the time of the snapshot. This is useful to create checkpoints in time that you may go back and reference, or use in a publication. Snapshots are immutable and cannot be changed. They may only be deleted.

Usage

```
synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_version=TRUE)
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
comment	(Optional[str]) A unique comment to associate with the snapshot.
label	(Optional[str]) A unique label to associate with the snapshot. If this is not a unique label an error will be raised when you store this to Synapse.
include_activity	(bool) If TRUE the activity will be included in snapshot if it exists. In order to include the activity, the activity must have already been stored in Synapse by using the \$activity attribute on the EntityView and calling the synStore() function on the EntityView instance. Adding an activity to a snapshot of a view is meant to capture the provenance of the data at the time of the snapshot. Defaults to TRUE.
associate_activity_to_new_version	(bool) If TRUE the activity will be associated with the new version of the view. If FALSE the activity will not be associated with the new version of the view. Defaults to TRUE.
timeout	(int) The number of seconds to wait for the job to complete or progress before raising a timeout error. Defaults to 120.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A TableUpdateTransaction object which includes the version number of the snapshot.

Examples

```
## Not run:
## Example 1: Creating a snapshot of a view with an activity
# Create a snapshot of a view and include the activity. The activity must
# have been stored in Synapse by using the $activity attribute on the EntityView
# and calling the synStore() function on the EntityView instance. Adding an activity
# to a snapshot of an EntityView is meant to capture the provenance of the data at
# the time of the snapshot.
library(synapser)
synLogin()

view = EntityView(id = "syn4567")
snapshot = view |> synSnapshot(
  label = "Q1 2025",
  comment = "Results collected in Lab A",
  include_activity = TRUE,
  associate_activity_to_new_version = TRUE
)
snapshot

## Example 2: Creating a snapshot of a view without an activity
# Create a snapshot of a view without including the activity. This is used in
# cases where we do not have any Provenance to associate with the snapshot and
# we do not want to persist any activity that may be present on the view to
# the new version of the view.
library(synapser)
synLogin()

view = EntityView(id = "syn4567")
snapshot = view |> synSnapshot(
  label = "Q1 2025",
  comment = "Results collected in Lab A",
  include_activity = FALSE,
  associate_activity_to_new_version = FALSE
)
snapshot

## End(Not run)
```

synSnapshot

SubmissionView : snapshot

Description

Creates a snapshot of the SubmissionView entity. Synapse handles snapshot creation differently for Table- and View-like entities. View snapshots are created using the asynchronous job API.

Making a snapshot of a view allows you to create an immutable version of the view at the time of the snapshot. This is useful to create checkpoints in time that you may go back and reference, or use in a publication. Snapshots are immutable and cannot be changed. They may only be deleted.

Usage

```
synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_version=TRUE)
```

Arguments

instance (SubmissionView) The SubmissionView instance to operate on.

comment (Optional[str]) A unique comment to associate with the snapshot.

label (Optional[str]) A unique label to associate with the snapshot. If this is not a unique label an error will be raised when you store this to Synapse.

include_activity (bool) If TRUE the activity will be included in snapshot if it exists. In order to include the activity, the activity must have already been stored in Synapse by using the \$activity attribute on the SubmissionView and calling the [synStore\(\)](#) function on the SubmissionView instance. Adding an activity to a snapshot of a table is meant to capture the provenance of the data at the time of the snapshot. Defaults to TRUE.

associate_activity_to_new_version (bool) If TRUE the activity will be associated with the new version of the table. If FALSE the activity will not be associated with the new version of the table. Defaults to TRUE.

timeout (int) The number of seconds to wait for the async job to complete. Defaults to 120.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A TableUpdateTransaction object which includes the version number of the snapshot.

Examples

```
## Not run:
## Example 1: Creating a snapshot of a view with an activity
# Create a snapshot of a view and include the activity. The activity must have
# been stored in Synapse by using the $activity attribute on the SubmissionView
# and calling the synStore() function on the SubmissionView instance. Adding an
# activity to a snapshot of a view is meant to capture the provenance of the
# data at the time of the snapshot.
library(synapser)
synLogin()

view = SubmissionView(id = "syn4567") |> synGet()
view$activity = Activity(name = "Activity for snapshot")
view |> synStore() # Store the activity

snapshot = view |> synSnapshot(
  label = "Q1 2025",
  comment = "Submissions reviewed in Lab A",
  include_activity = TRUE,
```



```
        associate_activity_to_new_version = TRUE
    )
    snapshot

## Example 2: Creating a snapshot of a view without an activity
# Create a snapshot of a view without including the activity. This is used in
# cases where we do not have any Provenance to associate with the snapshot and
# we do not want to persist any activity that may be present on the view to
# the new version of the view.
library(synapser)
synLogin()

view = SubmissionView(id = "syn4567") |> synGet()
snapshot = view |> synSnapshot(
  label = "Q1 2025",
  comment = "Submissions reviewed in Lab A",
  include_activity = FALSE,
  associate_activity_to_new_version = FALSE
)
snapshot

## End(Not run)
```

synSnapshot	<i>Table : snapshot</i>
-------------	-------------------------

Description

Request to create a new snapshot of a table. The provided comment, label, and activity will be applied to the current version thereby creating a snapshot and locking the current version. After the snapshot is created a new version will be started with an 'in-progress' label.

Usage

```
synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_version=TRUE)
```

Arguments

- instance (Table) The Table instance to operate on.
- comment (str) Comment to add to this snapshot to the table.
- label (str) Label to add to this snapshot to the table. The label must be unique, if a label is not provided a unique label will be generated.
- include_activity (bool) If TRUE the activity will be included in snapshot if it exists. In order to include the activity, the activity must have already been stored in Synapse by using the \$activity attribute on the Table and calling the [synStore\(\)](#) method on the Table instance. Adding an activity to a snapshot of a table is meant to capture the provenance of the data at the time of the snapshot.

`associate_activity_to_new_version`
 (bool) If TRUE the activity will be associated with the new version of the table.
 If FALSE the activity will not be associated with the new version of the table.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A named list that matches: <<https://rest-docs.synapse.org/rest/org/sagebionetworks/repo/model/table/SnapshotResponse.htm>

Examples

```
## Not run:
## Example 1: Creating a snapshot of a table
# Comment and label are optional, but filled in for this example.
library(synapser)
synLogin()

my_table = Table(id = "syn1234")
my_table |> synSnapshot(
  comment = "This is a new snapshot comment",
  label = "This is a unique label"
)

## Example 2: Including the activity (Provenance) in the snapshot and not pulling it forward to the new in-progress version
# By default this method is set up to include the activity in the snapshot and
# then pull the activity forward to the new version. If you do not want to
# include the activity in the snapshot you can set include_activity to
# FALSE. If you do not want to pull the activity forward to the new version
# you can set associate_activity_to_new_version to FALSE.

# See the $activity attribute on the Table class for more information on how
# to interact with the activity.
library(synapser)
synLogin()

my_table = Table(id = "syn1234")
my_table |> synSnapshot(
  comment = "This is a new snapshot comment",
  label = "This is a unique label",
  include_activity = TRUE,
  associate_activity_to_new_version = FALSE
)

## End(Not run)
```

Description

Starts an agent session.

Usage

synStart(instance, synapse_client=NULL)

Arguments

- instance (AgentSession) The AgentSession instance to operate on.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The new AgentSession object.

Examples

```
## Not run:
## Start a session and send a prompt.
# Start a session with a custom agent by providing the agent's registration ID
# and calling synStart(). Then, send a prompt to the agent.
library(synapser)
synLogin()

my_session = AgentSession(agent_registration_id = "foo") |> synStart()
my_session |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## End(Not run)
```

synStartSession	Agent : start_session
-----------------	-----------------------

Description

Starts an agent session. Adds the session to the Agent’s sessions named list and sets it as the current session.

Usage

synStartSession(instance, access_level="PUBLICLY_ACCESSIBLE", synapse_client=NULL)

Arguments

instance	(Agent) The Agent instance to operate on.
access_level	(Optional[AgentSessionAccessLevel]) The access level of the agent session. Must be one of PUBLICLY_ACCESSIBLE, READ_YOUR_PRIVATE_DATA, or WRITE_YOUR_PRIVATE_DATA. Defaults to PUBLICLY_ACCESSIBLE.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The new AgentSession object.

Examples

```
## Not run:
## Example 1: Start a session and send a prompt with the baseline Synapse Agent.
# The baseline Synapse Agent is the default agent used when a registration ID is not provided.
library(synapser)
synLogin()

my_agent = Agent()
my_agent |> synStartSession()
my_agent |> synPrompt(
  prompt = "Can you tell me about the AD Knowledge Portal dataset?",
  enable_trace = TRUE,
  print_response = TRUE
)

## Example 2: Start a session and send a prompt with a custom agent.
# If you have already registered a custom agent, you can start a session
# with it by providing its cloud agent ID.
library(synapser)
synLogin()

my_agent = Agent(cloud_agent_id = "foo")
my_agent |> synStartSession()
my_agent |> synPrompt(
  prompt = "Hello",
  enable_trace = TRUE,
  print_response = TRUE
)

## End(Not run)
```

Description

Factory method to store any Synapse entity.

This method serves as a unified interface for storing any type of Synapse entity. It automatically applies the appropriate store logic based on the entity type and the options provided.

Usage

```
synStore(entity, parent=NULL, file_options=NULL, container_options=NULL, table_options=NULL, json_sch
```

Arguments

entity	(Union[AgentSession, CurationTask, Dataset, DatasetCollection, EntityView, Evaluation, File, Folder, FormData, FormGroup, Grid, JSONSchema, Link, MaterializedView, Project, RecordSet, Organization, SearchIndex, SubmissionView, Table, Team, VirtualTable]) The entity instance to store. Must be one of the supported entity types.
parent	(Optional[Union[Folder, Project]]) The parent folder or project to store the entity in. Only applicable for File, Folder, Link, and RecordSet entities. Ignored for other entity types.
file_options	(Optional[StoreFileOptions]) StoreFileOptions to apply. Only applies to File and RecordSet entities. Ignored for other entity types.
container_options	(Optional[StoreContainerOptions]) StoreContainerOptions to apply. Only applies to Project and Folder entities. Ignored for other entity types.
table_options	(Optional[StoreTableOptions]) StoreTableOptions to apply. Only applies to Table-like entities (Table, Dataset, EntityView, MaterializedView, SubmissionView, VirtualTable, DatasetCollection). Ignored for other entity types.
json_schema_options	(Optional[StoreJSONSchemaOptions]) StoreJSONSchemaOptions to apply. Only applies to JSONSchema entities. Required for JSONSchema. Ignored for other entity types.
grid_options	(Optional[StoreGridOptions]) StoreGridOptions to apply. Only applies to Grid entities. Ignored for other entity types.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The stored Synapse entity model instance.

Errors

- An error is raised if the entity is not a supported type.

Examples

```
## Not run:
## Example 1: Storing a file
# Store a file to Synapse:
library(synapser)
synLogin()

file = File(path = "/path/to/file.txt", parent_id = "syn123456")
stored_file = file |> synStore()
print(sprintf("Stored file: %s (ID: %s)", stored_file$name, stored_file$id))

## Example 2: Storing a file with custom options
# Store a file with custom options:
library(synapser)
synLogin()

file = File(path = "/path/to/file.txt", parent_id = "syn123456")
stored_file = file |> synStore(
  file_options = StoreFileOptions(
    synapse_store = TRUE,
    content_type = "text/plain",
    merge_existing_annotations = TRUE
  )
)
print(sprintf("Stored file: %s", stored_file$name))

## Example 3: Storing a folder
# Store a folder to Synapse:
library(synapser)
synLogin()

folder = Folder(name = "My Folder", parent_id = "syn123456")
stored_folder = folder |> synStore()
print(sprintf("Stored folder: %s (ID: %s)", stored_folder$name, stored_folder$id))

## Example 4: Storing a project
# Store a project to Synapse:
library(synapser)
synLogin()

project = Project(name = "My Project")
stored_project = project |> synStore()
print(sprintf("Stored project: %s (ID: %s)", stored_project$name, stored_project$id))

## Example 5: Storing a table
# Store a table to Synapse:
library(synapser)
synLogin()

table = Table(
  name = "My Table",
  parent_id = "syn123456",
```

```

        columns = list(Column(name = "col1", column_type = "STRING", maximum_size = 50L))
    )
    stored_table = table |> synStore(
        table_options = StoreTableOptions(dry_run = FALSE, job_timeout = 600L)
    )
    print(sprintf("Stored table: %s (ID: %s)", stored_table$name, stored_table$id))

## Example 6: Storing a link
# Store a link to Synapse:
library(synapser)
synLogin()

link = Link(name = "My Link", parent_id = "syn123456", target_id = "syn789")
stored_link = link |> synStore()
print(sprintf("Stored link: %s (ID: %s)", stored_link$name, stored_link$id))

## Example 7: Storing with parent parameter
# Store an entity by passing the parent as a parameter:
library(synapser)
synLogin()

parent_folder = Folder(id = "syn123456")
file = File(path = "/path/to/file.txt")
stored_file = file |> synStore(parent = parent_folder)
print(sprintf("Stored file: %s in folder %s", stored_file$name, parent_folder$id))

## Example 8: Storing an Evaluation
# Store an evaluation (challenge queue):
library(synapser)
synLogin()

evaluation = Evaluation(
    name = "My Challenge Evaluation",
    description = "Evaluation for my data challenge",
    content_source = "syn123456"
)
stored_evaluation = evaluation |> synStore()
print(sprintf("Stored evaluation: %s (ID: %s)", stored_evaluation$name, stored_evaluation$id))

## Example 9: Creating a Team
# Create a new team (note: Teams use create, not update):
library(synapser)
synLogin()

team = Team(
    name = "My Research Team",
    description = "A team for collaborative research"
)
created_team = team |> synStore()
print(sprintf("Created team: %s (ID: %s)", created_team$name, created_team$id))

## End(Not run)

```

synStoreRows

*Table : store_rows***Description**

Add or update rows in Synapse from the sources defined below. In most cases the result of this function call will append rows to the table. In the case of an update this method works on a full row replacement. What this means is that you may not do a partial update of a row. If you want to update a row you must pass in all the data for that row, or the data for the columns not provided will be set to null.

If you'd like to update a row see the example [Updating rows in a table](#) below.

If you'd like to perform an `upsert` or partial update of a row you may use the [synUpsertRows\(\)](#) function. See that function for more information.

Note the following behavior for the order of columns:

- If a column is added via the [synAddColumn\(\)](#) function it will be added at the index you specify, or at the end of the columns list.
- If column(s) are added during the construction of your Table instance, ie. `Table(columns = list(Column(name = "foo")))`, they will be added at the beginning of the columns list.
- If you use the [synStoreRows\(\)](#) function and the `schema_storage_strategy` is set to `INFER_FROM_DATA` the columns will be added at the end of the columns list.

How datetime values are interpreted:

Synapse DATE columns store an exact moment in time, represented as the number of milliseconds since 1970-01-01 00:00:00 UTC. Before upload, datetime values are converted to that representation as follows:

- Timezone-aware datetimes (POSIXct values created with a tz, e.g. `as.POSIXct("2017-02-14 11:23", tz = "America/Los_Angeles")`) are converted to UTC exactly. This is the recommended way to pass datetime data: the stored value does not depend on the timezone settings of the machine performing the upload or on the date the upload is run.
- Naive datetimes (POSIXct values without a tz) are assumed to be in the local timezone of the machine **at the time of upload**. That single UTC offset is applied to every value, including values whose dates fall in a different daylight saving period, which will be stored shifted by one hour. For example, uploading 2017-02-14 11:23 (a PST date, UTC-8) from a machine currently on PDT (UTC-7) stores 2017-02-14 18:23 UTC instead of the correct 19:23 UTC. To avoid this, create your datetimes with a timezone, e.g. `as.POSIXct(x, tz = "America/Los_Angeles")`.
- Plain R Date values (as opposed to datetimes) are treated as midnight of that date and converted the same way naive datetimes are: using the local timezone of the machine **at the time of upload**, with the same daylight-saving-shift risk described above.
- Midnight values deserve extra care: the one-hour daylight saving shift described above can move a naive midnight datetime — or a Date, which is treated as midnight — to 11 PM of the previous day, changing the calendar date the value displays as.

Why timezone-aware values are recommended:

A naive datetime such as 2017-02-14 11:23 is only "what a clock on the wall said." Before it can be stored as an exact moment in time, something has to answer: *a clock where?* Timezone-aware

values answer that question in the data itself; naive values leave the client to guess, and it guesses the uploading machine's current timezone.

A zone name like "America/Los_Angeles" does not mean a fixed offset such as UTC-8. It means "a Los Angeles wall clock" — and those clocks move twice a year: UTC-7 in summer (PDT), UTC-8 in winter (PST). When you create datetimes with `as.POSIXct(x, tz = "America/Los_Angeles")`, R looks up what LA clocks were set to on each value's own date:

- 2017-02-14 → winter → interpreted using UTC-8
- 2018-10-01 → summer → interpreted using UTC-7

When the data is queried back with `synQuery(..., convert_to_datetime = TRUE)`, DATE columns are returned as datetimes in UTC. See vignette("tables", package = "synapser") for how to work with the returned values and view them in another timezone.

Limitations:

- Synapse limits the number of rows that may be stored in a single request to a CSV file that is 1GB. If you are storing a CSV file that is larger than this limit the data will be chunked into smaller requests. This process is done by reading the file once to determine what the row and byte boundaries are and calculating the MD5 hash of that portion, then reading the file again to send the data to Synapse. This process is done to ensure that the data is not corrupted during the upload process, in addition Synapse requires the MD5 hash of the data to be sent in the request along with the number of bytes that are being sent.
- The limit of 1GB is also enforced when storing a named list or a data frame. The data will be converted to a CSV format using the `to_csv()` pandas function. If you are storing more than a 1GB file it is recommended that you store the data as a CSV and use the file path to upload the data. This is due to the fact that the data frame chunking process is slower than reading portions of a file on disk and calculating the MD5 hash of that portion.

The following is a Sequence Diagram that describes the process noted in the limitation above. It shows how the data is chunked into smaller requests when the data exceeds the limit of 1GB, and how portions of the data are read from the CSV file on disk while being uploaded to Synapse.

```
sequenceDiagram
    participant User
    participant Table
    participant FileSystem
    participant Synapse

    User->>Table: store_rows(values)

    alt CSV size > 1GB
        Table->>Synapse: Apply schema changes before uploading
        note over Table, FileSystem: Read CSV twice
        Table->>FileSystem: Read entire CSV (First Pass)
        FileSystem-->>Table: Compute chunk sizes & MD5 hashes

        loop Read and Upload CSV chunks (Second Pass)
            Table->>FileSystem: Read next chunk from CSV
            FileSystem-->>Table: Return bytes
            Table->>Synapse: Upload CSV chunk
            Synapse-->>Table: Return file_handle_id
        end
    end
```

```

        Table->>Synapse: Send 'TableUpdateTransaction' to append/update rows
        Synapse-->>Table: Transaction result
    end
else
    Table->>Synapse: Upload CSV without splitting & Any additional schema changes
    Synapse-->>Table: Return file_handle_id
    Table->>Synapse: Send 'TableUpdateTransaction' to append/update rows
    Synapse-->>Table: Transaction result
end

Table-->>User: Upload complete

```

The following is a Sequence Diagram that describes the process noted in the limitation above for data frames. It shows how the data is chunked into smaller requests when the data exceeds the limit of 1GB, and how portions of the data are read from the data frame while being uploaded to Synapse.

```

sequenceDiagram
    participant User
    participant Table
    participant MemoryBuffer
    participant Synapse

    User->>Table: store_rows(DataFrame)

    loop For all rows in DataFrame in 100 row increments
        Table->>MemoryBuffer: Convert DataFrame rows to CSV in-memory
        MemoryBuffer-->>Table: Compute chunk sizes & MD5 hashes
    end

    alt Multiple chunks detected
        Table->>Synapse: Apply schema changes before uploading
    end

    loop For all chunks found in first loop
        loop for all parts in chunk byte boundry
            Table->>MemoryBuffer: Read small (< 8MB) part of the chunk
            MemoryBuffer-->>Table: Return bytes (with correct offset)
            Table->>Synapse: Upload part
            Synapse-->>Table: Upload response
        end
        Table->>Synapse: Complete upload
        Synapse-->>Table: Return file_handle_id
        Table->>Synapse: Send 'TableUpdateTransaction' to append/update rows
        Synapse-->>Table: Transaction result
    end

    Table-->>User: Upload complete

```

Usage

```
synStoreRows(instance, values, schema_storage_strategy=NULL, column_expansion_strategy=NULL, dry_run=
```

Arguments

instance (Table) The Table instance to operate on.

values (Union[str, dict[str, Any], object]) Supports storing data from the following sources:

- A string holding the path to a CSV file. If the `schema_storage_strategy` is set to `NULL` the data will be uploaded as is. If `schema_storage_strategy` is set to `INFER_FROM_DATA` the data will be read into a **Pandas DataFrame**. The code makes assumptions about the format of the columns in the CSV as detailed in the `csv_to_pandas_df` function. You may pass in additional arguments to the `csv_to_pandas_df` function by passing them in as keyword arguments to this function.

- A named list where the key is the column name and the value is one or more values. The values will be wrapped into a **Pandas DataFrame**. You may pass in additional arguments to the pandas DataFrame constructor by passing them in as keyword arguments to this function. Read about the available arguments in the **Pandas DataFrame** documentation.

- A **Pandas DataFrame**

schema_storage_strategy (SchemaStorageStrategy) Determines how to automate the creation of columns based on the data that is being stored. If you want to have full control over the schema you may set this to `NULL` and create the columns manually.

The limitation with this behavior is that the columns created may only be of the following types:

- STRING
- LARGETEXT
- INTEGER
- DOUBLE
- BOOLEAN
- DATE

The determination is based on how this pandas function infers the data type: `infer_dtype`

This may also only set the `name`, `column_type`, and `maximum_size` of the column when the column is created. If this is used to update the column the `maximum_size` will only be updated depending on the value of `column_expansion_strategy`. The other attributes of the column will be set to the default values on create, or remain the same if the column already exists.

The usage of this feature will never delete a column, shrink a column, or change

the type of a column that already exists. If you need to change any of these attributes you must do so after getting the table via a `synGet()` call, updating the columns as needed, then calling `synStore()` on the table.

<code>column_expansion_strategy</code>	(ColumnExpansionStrategy) Determines how to automate the expansion of columns based on the data that is being stored. The options given allow cells with a limit on the length of content (Such as strings) to be expanded to a larger size if the data being stored exceeds the limit. If you want to have full control over the schema you may set this to NULL and create the columns manually. String type columns are the only ones that support this feature.
<code>dry_run</code>	(bool) Log the actions that would be taken, but do not actually perform the actions. This will not print out the data that would be stored or modified as a result of this action. It will print out the actions that would be taken, such as creating a new column, updating a column, or updating table metadata. This is useful for debugging and understanding what actions would be taken without actually performing them.
<code>additional_changes</code>	(list[Union[TableSchemaChangeRequest, UploadToTableRequest, AppendableRowSetRequest]]) Additional changes to the table that should execute within the same transaction as appending or updating rows. This is used as a part of the <code>synUpsertRows()</code> function call to allow for the updating of rows and the updating of the table schema in the same transaction. In most cases you will not need to use this argument.
<code>insert_size_bytes</code>	(int) The maximum size of data that will be stored to Synapse within a single transaction. The API has a limit of 1GB, but the default is set to 900 MB to allow for some overhead in the request. The implication of this limit is that when you are storing a CSV that is larger than this limit the data will be chunked into smaller requests by reading the file once to determine what the row and byte boundaries are and calculating the MD5 hash of that portion, then reading the file again to send the data to Synapse. This process is done to ensure that the data is not corrupted during the upload process, in addition Synapse requires the MD5 hash of the data to be sent in the request along with the number of bytes that are being sent. This argument is also used when storing a named list or a data frame. The data will be converted to a CSV format using the <code>to_csv()</code> pandas function. When storing data as a data frame the minimum that it will be chunked to is 100 rows of data, regardless of if the data is larger than the limit.
<code>csv_table_descriptor</code>	(Optional[CsvTableDescriptor]) When passing in a CSV file this will allow you to specify the format of the CSV file. This is only used when the <code>values</code> argument is a string holding the path to a CSV file. See CsvTableDescriptor for more information.
<code>date_columns</code>	(Optional[list[character]]) (CSV file only) The names of columns in your CSV file that contain dates or datetimes stored as formatted strings (e.g. "2024-01-15" or "01/15/2024 13:30"). The columns are parsed with <code>pandas.to_datetime</code> and converted to epoch time in milliseconds before the data is uploaded, which is the format Synapse requires for DATE columns. The conversion is done by

reading the CSV file into a pandas DataFrame and uploading a temporary copy with the converted values, which requires the CSV file to contain a header row. When `schema_storage_strategy` is set to `INFER_FROM_DATA` the parsed columns will be inferred as DATE columns. The parsed values are naive date-times unless the strings carry a UTC offset — see *How datetime values are interpreted* above for how they are converted to epoch time. When the offsets in a column differ between rows (e.g. a mix of `-0800` and `-0700` values) the values are normalized to UTC, preserving the exact moments in time.

date_format (Optional[character or named list]) (CSV file only) How the strings in `date_columns` are formatted — a **strftime format string** (e.g. `"%m/%d/%Y"`) applied to every column, or a named list mapping column names to their formats. Supply this so that ambiguous dates (e.g. `"01/02/2024"`) are not silently misinterpreted and to optimize the data upload performance. If the values in a column do not match the format an error is raised.

read_csv_kwargs (Optional[dict[str, Any]]) Additional arguments to pass to the `pd.read_csv` function when reading in a CSV file. This is only used when the values argument is a string holding the path to a CSV file and either `schema_storage_strategy` is set to `INFER_FROM_DATA` or `date_columns` is provided. See <https://pandas.pydata.org/docs/reference/> for complete list of supported arguments.

to_csv_kwargs (Optional[dict[str, Any]]) Additional arguments to pass to the `pd.DataFrame.to_csv` function when writing the data to a CSV file. This is only used when the values argument is a Pandas DataFrame. See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html for complete list of supported arguments. Any kwargs you supply are merged on top of the default of using a backslash as the escape character, so you only need to override `escapechar` explicitly if you want different behavior.

job_timeout (int) The maximum amount of time to wait for a job to complete. This is used when inserting, and updating rows of data. Each individual request to Synapse will be sent as an independent job. If the timeout is reached a timeout error will be raised. The default is 600 seconds

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Example 1: Inserting rows into a table that already has columns
# This example shows how you may insert rows into a table.

# Suppose we have a table with the following columns:

# | col1 | col2 | col3 |
# |-----|-----| ----|

# The following code will insert rows into the table:
```

```

library(synapser)
synLogin()

# Also works with Dataset
data_to_insert = data.frame(
  col1 = c("A", "B", "C"),
  col2 = c(1, 2, 3),
  col3 = c(1, 2, 3)
)

Table(id = "syn1234") |> synStoreRows(values = data_to_insert)

# The resulting table will look like this:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 1    | 1    |
# | B    | 2    | 2    |
# | C    | 3    | 3    |

## Example 2: Inserting rows into a table that does not have columns
# This example shows how you may insert rows into a table that does not have
# columns. The columns will be inferred from the data that is being stored.
library(synapser)
synLogin()

# Also works with Dataset
data_to_insert = data.frame(
  col1 = c("A", "B", "C"),
  col2 = c(1, 2, 3),
  col3 = c(1, 2, 3)
)

Table(id = "syn1234") |> synStoreRows(
  values = data_to_insert,
  schema_storage_strategy = "INFER_FROM_DATA"
)

# The resulting table will look like this:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 1    | 1    |
# | B    | 2    | 2    |
# | C    | 3    | 3    |

## Example 3: Using the dry_run option with a SchemaStorageStrategy of INFER_FROM_DATA
# This example shows how you may use the dry_run option with the
# SchemaStorageStrategy set to INFER_FROM_DATA. This will show you the
# actions that would be taken, but not actually perform the actions.
library(synapser)
synLogin()

```

```

# Also works with Dataset
data_to_insert = data.frame(
  col1 = c("A", "B", "C"),
  col2 = c(1, 2, 3),
  col3 = c(1, 2, 3)
)

Table(id = "syn1234") |> synStoreRows(
  values = data_to_insert,
  dry_run = TRUE,
  schema_storage_strategy = "INFER_FROM_DATA"
)

# The result of running this action will print to the console the actions that
# would be taken, but not actually perform the actions.

## Example 4: Updating rows in a table
# This example shows how you may query for data in a table, update the data,
# and then store the updated rows back in Synapse.

# Suppose we have a table that has the following data:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 1    | 1    |
# | B    | 2    | 2    |
# | C    | 3    | 3    |

# Behind the scenes the table also has ROW_ID and ROW_VERSION columns
# which are used to identify the row that is being updated. These columns
# are not shown in the table above, but are included in the data that is
# returned when querying the table. If you add data that does not have these
# columns the data will be treated as new rows to be inserted.
library(synapser)
synLogin()

# Also works with Dataset
query_results = synQuery(query = "select * from syn1234 where col1 in ('A', 'B')")

# Update col2 of the row where col1 is A to 22
query_results[query_results$col1 == "A", "col2"] = 22

# Update col3 of the row where col1 is B to 33
query_results[query_results$col1 == "B", "col3"] = 33

Table(id = "syn1234") |> synStoreRows(values = query_results)

# The resulting table will look like this:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 22   | 1    |
# | B    | 2    | 33   |

```

```

# | C      | 3      | 3      |

## Example 5: Inserting rows from a CSV file that contains date columns
# This example shows how you may insert rows from a CSV file that contains
# date columns. The date columns are converted
# to epoch time in milliseconds before the data is uploaded so that Synapse
# can store them in DATE columns.

# Suppose we have a CSV file with the following data:

# | col1 | date_col1 | date_col2 |
# |-----|-----|-----|
# | A     | 01/15/2024 | 2024-01-20 |
# | B     | 02/20/2024 | 2024-02-25 |
library(synapser)
synLogin()

Table(id = "syn1234") |> synStoreRows(
  values = "path/to/file.csv",
  schema_storage_strategy = "INFER_FROM_DATA",
  date_columns = list("date_col1", "date_col2"),
  date_format = list(date_col1 = "%m/%d/%Y", date_col2 = "%Y-%m-%d")
)

## End(Not run)

```

synSyncFromSynapse *Folder : sync_from_synapse*

Description

Sync this container and all possible sub-folders from Synapse. By default this will download the files that are found and it will populate the files and folders attributes with the found files and folders, along with all other entity types (tables, entityviews, etc.) present in the container. If you only want to retrieve the full tree of metadata about your container specify `download_file` as `FALSE`.

This works similar to the Python `synapseutils.syncFromSynapse` utility function, and generates a `manifest.csv` file in each synced directory. The manifest uses CSV format with `parentId` and `ID` columns, interoperable with the Synapse UI download cart and `synDownloadListFiles()` output.

Supports syncing Files, Folders, Tables, EntityViews, SubmissionViews, Datasets, DatasetCollections, MaterializedViews, and VirtualTables from Synapse. The metadata for these entity types will be populated in their respective attributes (`files`, `folders`, `tables`, `entityviews`, `submissionviews`, `datasets`, `datasetcollections`, `materializedviews`, `virtualtables`) if they are found within the container.

Usage

```
synSyncFromSynapse(instance, path=NULL, recursive=TRUE, download_file=TRUE, if_collision="overwrite...)
```


Arguments

instance	(Folder) The Folder instance to operate on.
path	(Optional[str]) An optional path where the file hierarchy will be reproduced. If not specified the files will by default be placed in the synapseCache.
recursive	(bool) Whether or not to recursively get the entire hierarchy of the folder and sub-folders.
download_file	(bool) Whether to download the files found or not.
if_collision	(str) Determines how to handle file collisions. May be <ul style="list-style-type: none"> - overwrite.local - keep.local - keep.both
failure_strategy	(str) Determines how to handle failures when retrieving children under this Folder and an exception occurs. One of "LOG_EXCEPTION" or "RAISE_EXCEPTION".
include_activity	(bool) Whether to include the activity of the files.
follow_link	(bool) Whether to follow a link entity or not. Links can be used to point at other Synapse entities.
link_hops	(int) The number of hops to follow the link. A number of 1 is used to prevent circular references. There is nothing in place to prevent infinite loops. Be careful if setting this above 1.
queue	(Queue) An optional queue to use to download files in parallel.
include_types	(Optional[list[str]]) Must be a list of entity types (ie. list("folder", "file")) which can be found here . Defaults to list("folder", "file", "table", "entityview", "dockerrepo", "submissionview", "dataset", "datasetcollection", "materializedview", "virtualtable", "searchindex").
manifest	(character) Determines whether to generate a manifest CSV file. Options are: <ul style="list-style-type: none"> - "all" (default): generate manifest.csv in every synced directory - "root": generate manifest.csv only in the root path directory - "suppress": do not generate any manifest file
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The object that was called on. This will be the same object that was called on to start the sync.

Errors

An error is raised if the folder does not have an id set.

A sequence diagram for this method is as follows:

```

sequenceDiagram
    autonumber
    participant project_or_folder
    activate project_or_folder
    project_or_folder->>sync_from_synapse: Recursive search and download files
    activate sync_from_synapse
        opt Current instance not retrieved from Synapse
            sync_from_synapse->>project_or_folder: Call .get() method
            project_or_folder-->>sync_from_synapse: .
        end

        loop For each return of the generator
            sync_from_synapse->>client: call .getChildren() method
            client-->>sync_from_synapse: .
            note over sync_from_synapse: Append to a running list
        end

        loop For each child
            note over sync_from_synapse: Create all pending_tasks at current depth

            alt Child is File
                note over sync_from_synapse: Append file.get() method
            else Child is Folder
                note over sync_from_synapse: Append folder.get() method
                alt Recursive is True
                    note over sync_from_synapse: Append folder.sync_from_synapse() method
                end
            else Child is Link and hops > 0
                note over sync_from_synapse: Append task to follow link
            end
        end

        loop For each task in pending_tasks
            par file.get()
                sync_from_synapse->>File: Retrieve File metadata and Optionally download
                File->>client: .get()
                client-->>File: .
                File-->>sync_from_synapse: .
            and folder.get()
                sync_from_synapse->>Folder: Retrieve Folder metadata
                Folder->>client: .get()
                client-->>Folder: .
                Folder-->>sync_from_synapse: .
            and folder.sync_from_synapse_async()
            note over sync_from_synapse: This is a recursive call to sync_from_synapse
            sync_from_synapse->>sync_from_synapse: Recursive call to .sync_from_synapse_async()
            and _follow_link
                sync_from_synapse ->>client: call get_entity_id_bundle2 function
            end
        end
    end

```

```

        client-->sync_from_synapse: .
        note over sync_from_synapse: Do nothing if not link
        note over sync_from_synapse: call _create_task_for_child and execute
        end
    end

    opt manifest != "suppress" and path is set
        alt manifest == "all"
            loop For each directory path
                sync_from_synapse->>manifest: call generate_manifest_csv(files, dir_path, syn)
                manifest-->>sync_from_synapse: manifest.csv written to dir_path
            end
        else manifest == "root"
            sync_from_synapse->>manifest: call generate_manifest_csv(all_files, root_path, syn)
            manifest-->>sync_from_synapse: manifest.csv written to root_path
        end
    end

    deactivate sync_from_synapse
    deactivate project_or_folder

```

Examples

```

## Not run:
## Using this function
# Suppose I want to walk the immediate children of a folder without downloading the files:
library(synapser)
synLogin()

my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(download_file = FALSE, recursive = FALSE)

for (folder in my_folder$folders) {
  print(folder$name)
}

for (file in my_folder$files) {
  print(file$name)
}

for (table in my_folder$tables) {
  print(table$name)
}

for (dataset in my_folder$datasets) {
  print(dataset$name)
}

# Suppose I want to download the immediate children of a folder:
library(synapser)
synLogin()

```

```

my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(path = "/path/to/folder", recursive = FALSE)

for (folder in my_folder$folders) {
  print(folder$name)
}

for (file in my_folder$files) {
  print(file$name)
}

# Suppose I want to sync only specific entity types from a Project:
library(synapser)
synLogin()

my_project = Project(id = "syn12345")
my_project = my_project |> synSyncFromSynapse(
  path = "/path/to/folder",
  include_types = list("folder", "file", "table", "dataset")
)

# Access different entity types
for (table in my_project$tables) {
  cat(sprintf("Table: %s\n", table$name))
}

for (dataset in my_project$datasets) {
  cat(sprintf("Dataset: %s\n", dataset$name))
}

# Suppose I want to download all the children of a Project and all sub-folders and files:
library(synapser)
synLogin()

my_project = Project(id = "syn12345")
my_project = my_project |> synSyncFromSynapse(path = "/path/to/folder")

# Suppose I want to download all the children of a Folder and all sub-folders and files and generate a manifest file:
library(synapser)
synLogin()

my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(path = "/path/to/folder", manifest = "all")

# Suppose I want to download a manifest file at the root path:
library(synapser)
synLogin()

my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(path = "/path/to/folder", manifest = "root", download_file = FALSE)

## End(Not run)

```

synSyncFromSynapse	<i>Project : sync_from_synapse</i>
--------------------	------------------------------------

Description

Sync this container and all possible sub-folders from Synapse. By default this will download the files that are found and it will populate the files and folders attributes with the found files and folders, along with all other entity types (tables, entityviews, etc.) present in the container. If you only want to retrieve the full tree of metadata about your container specify download_file as FALSE.

This works similarly to the legacy `synapseutils.syncFromSynapse` utility function, and generates a manifest.csv file in each synced directory. The manifest uses CSV format with parentId and ID columns, interoperable with the Synapse UI download cart and `synDownloadListFiles()` output.

Supports syncing Files, Folders, Tables, EntityViews, SubmissionViews, Datasets, DatasetCollections, MaterializedViews, and VirtualTables from Synapse. The metadata for these entity types will be populated in their respective attributes (files, folders, tables, entityviews, submissionviews, datasets, datasetcollections, materializedviews, virtualtables) if they are found within the container.

Usage

```
synSyncFromSynapse(instance, path=NULL, recursive=TRUE, download_file=TRUE, if_collision="overwrite...)
```

Arguments

instance	(Project) The Project instance to operate on.
path	(Optional[str]) An optional path where the file hierarchy will be reproduced. If not specified the files will by default be placed in the synapseCache.
recursive	(bool) Whether or not to recursively get the entire hierarchy of the folder and sub-folders.
download_file	(bool) Whether to download the files found or not.
if_collision	(str) Determines how to handle file collisions. May be: <ul style="list-style-type: none"> - overwrite.local - keep.local - keep.both
failure_strategy	(str) Determines how to handle failures when retrieving children under this Folder and an exception occurs. One of "LOG_EXCEPTION" or "RAISE_EXCEPTION".
include_activity	(bool) Whether to include the activity of the files.
follow_link	(bool) Whether to follow a link entity or not. Links can be used to point at other Synapse entities.

link_hops	(int) The number of hops to follow the link. A number of 1 is used to prevent circular references. There is nothing in place to prevent infinite loops. Be careful if setting this above 1.
queue	(Queue) An optional queue to use to download files in parallel.
include_types	(Optional[list[str]]) Must be a list of entity types (ie. list("folder", "file")) which can be found here . Defaults to list("folder", "file", "table", "entityview", "dockerrepo", "submissionview", "dataset", "datasetcollection", "materializedview", "virtualtable", "searchindex").
manifest	(character) Determines whether to generate a manifest CSV file. Options are: <ul style="list-style-type: none"> - "all" (default): generate manifest.csv in every synced directory - "root": generate manifest.csv only in the root path directory - "suppress": do not generate any manifest file
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The object that was called on. This will be the same object that was called on to start the sync.

Errors

An error is raised if the folder does not have an id set.

A sequence diagram for this method is as follows:

```
sequenceDiagram
    autonumber
    participant project_or_folder
    activate project_or_folder
    project_or_folder->>sync_from_synapse: Recursive search and download files
    activate sync_from_synapse
    opt Current instance not retrieved from Synapse
        sync_from_synapse->>project_or_folder: Call .get() method
        project_or_folder-->>sync_from_synapse: .
    end
    end

    loop For each return of the generator
        sync_from_synapse->>client: call .getChildren() method
        client-->>sync_from_synapse: .
        note over sync_from_synapse: Append to a running list
    end

    loop For each child
        note over sync_from_synapse: Create all pending_tasks at current depth

        alt Child is File
            note over sync_from_synapse: Append file.get() method
```

```

    else Child is Folder
        note over sync_from_synapse: Append folder.get() method
        alt Recursive is True
            note over sync_from_synapse: Append folder.sync_from_synapse() method
        end
    else Child is Link and hops > 0
        note over sync_from_synapse: Append task to follow link
    end
end

loop For each task in pending_tasks
    par file.get()
    sync_from_synapse->>File: Retrieve File metadata and Optionally download
        File->>client: .get()
        client-->>File: .
        File-->>sync_from_synapse: .
    and folder.get()
        sync_from_synapse->>Folder: Retrieve Folder metadata
        Folder->>client: .get()
        client-->>Folder: .
        Folder-->>sync_from_synapse: .
    and folder.sync_from_synapse_async()
    note over sync_from_synapse: This is a recursive call to sync_from_synapse
    sync_from_synapse->>sync_from_synapse: Recursive call to .sync_from_synapse_async()
    and _follow_link
        sync_from_synapse ->>client: call get_entity_id_bundle2 function
        client-->>sync_from_synapse: .
        note over sync_from_synapse: Do nothing if not link
        note over sync_from_synapse: call _create_task_for_child and execute
    end
end

opt manifest != "suppress" and path is set
    alt manifest == "all"
        loop For each directory path
            sync_from_synapse->>manifest: call generate_manifest_csv(files, dir_path, syn)
            manifest-->>sync_from_synapse: manifest.csv written to dir_path
        end
    else manifest == "root"
        sync_from_synapse->>manifest: call generate_manifest_csv(all_files, root_path, syn)
        manifest-->>sync_from_synapse: manifest.csv written to root_path
    end
end

deactivate sync_from_synapse
deactivate project_or_folder

```

Examples

```
## Not run:
## Using this function
# Suppose I want to walk the immediate children of a folder without downloading the files:
library(synapser)
synLogin()

my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(download_file = FALSE, recursive = FALSE)

for (folder in my_folder$folders) {
  cat(folder$name, "\n")
}

for (file in my_folder$files) {
  cat(file$name, "\n")
}

for (table in my_folder$tables) {
  cat(table$name, "\n")
}

for (dataset in my_folder$datasets) {
  cat(dataset$name, "\n")
}

# Suppose I want to download the immediate children of a folder:
my_folder = Folder(id = "syn12345")
my_folder = my_folder |> synSyncFromSynapse(path = "/path/to/folder", recursive = FALSE)

for (folder in my_folder$folders) {
  cat(folder$name, "\n")
}

for (file in my_folder$files) {
  cat(file$name, "\n")
}

# Suppose I want to sync only specific entity types from a Project:
my_project = Project(id = "syn12345")
my_project = my_project |> synSyncFromSynapse(
  path = "/path/to/folder",
  include_types = list("folder", "file", "table", "dataset")
)

# Access different entity types
for (table in my_project$tables) {
  cat(sprintf("Table: %s\n", table$name))
}

for (dataset in my_project$datasets) {
  cat(sprintf("Dataset: %s\n", dataset$name))
}
```



```

    }

    # Suppose I want to download all the children of a Project and all sub-folders and files:
    my_project = Project(id = "syn12345")
    my_project = my_project |> synSyncFromSynapse(path = "/path/to/folder")

    # Suppose I want to download all the children of a Project and all sub-folders and files and generate a manifest file:
    my_project = Project(id = "syn12345")
    my_project = my_project |> synSyncFromSynapse(path = "/path/to/folder", manifest = "all")

    # Suppose I want to download a manifest file at the root path:
    my_project = Project(id = "syn12345")
    my_project = my_project |> synSyncFromSynapse(path = "/path/to/folder", manifest = "root", download_file = FALSE)

## End(Not run)

```

synSyncToSynapse	<i>Folder : syn_to_synapse</i>
------------------	--------------------------------

Description

Upload files to Synapse using a manifest CSV file.

Accepts manifests produced by [synSyncFromSynapse\(\)](#), [synDownloadListFiles\(\)](#), or the Synapse UI download cart. The manifest must have at minimum a path and parentId column. All other columns that are not part of the standard manifest column set are treated as file annotations.

Standard manifest columns: ID, name, parentId, contentType, path, synapseStore, activityName, activityDescription, forceVersion, used, executed

Usage

```
synSyncToSynapse(instance, manifest_path, dry_run=FALSE, send_messages=TRUE, retries=4L, merge_existing=FALSE)
```

Arguments

instance	(Folder) The Folder instance to operate on.
manifest_path	(character) Path to the CSV manifest file.
dry_run	(logical) If TRUE, perform full validation of the manifest (including verifying that all parent containers exist in Synapse) but skip the actual file upload.
send_messages	(logical) If TRUE, send a Synapse notification message on completion.
retries	(integer) Number of notification retries (only relevant when send_messages=TRUE).
merge_existing_annotations	(logical) If TRUE, merge manifest annotations with existing annotations on Synapse. If FALSE, overwrite them.
associate_activity_to_new_version	(logical) If TRUE and a version update occurs, the existing Synapse activity is associated with the new version.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

List of File entities that were created or updated. Returns an empty list if dry_run=TRUE or if no rows were eligible for upload.

Examples

```
## Not run:
## Using this function
library(synapser)
synLogin()

folder = Folder(id = "syn12345")
uploaded = folder |> synSyncToSynapse(manifest_path = "/path/to/manifest.csv")

## End(Not run)
```

synSyncToSynapse	<i>Project : sync_to_synapse</i>
------------------	----------------------------------

Description

Upload files to Synapse using a manifest CSV file.

Accepts manifests produced by [synSyncFromSynapse\(\)](#), [synDownloadListFiles\(\)](#), or the Synapse UI download cart. The manifest must have at minimum a path and parentId column. All other columns that are not part of the standard manifest column set are treated as file annotations.

Standard manifest columns: ID, name, parentId, contentType, path, synapseStore, activityName, activityDescription, forceVersion, used, executed

Usage

```
synSyncToSynapse(instance, manifest_path, dry_run=FALSE, send_messages=TRUE, retries=4L, merge_existi
```

Arguments

- instance (Project) The Project instance to operate on.
- manifest_path (character) Path to the CSV manifest file.
- dry_run (logical) If TRUE, perform full validation of the manifest (including verifying that all parent containers exist in Synapse) but skip the actual file upload.
- send_messages (logical) If TRUE, send a Synapse notification message on completion.
- retries (integer) Number of notification retries (only relevant when send_messages=TRUE).
- merge_existing_annotations (logical) If TRUE, merge manifest annotations with existing annotations on Synapse. If FALSE, overwrite them.
- associate_activity_to_new_version (logical) If TRUE and a version update occurs, the existing Synapse activity is associated with the new version.
- synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

List of File entities that were created or updated. Returns an empty list if dry_run=TRUE or if no rows were eligible for upload.

Examples

```
## Not run:
## Using this function
library(synapser)
synLogin()

project = Project(id = "syn12345")
uploaded = project |> synSyncToSynapse(manifest_path = "/path/to/manifest.csv")

## End(Not run)
```

synUnbindSchema	<i>EntityView : unbind_schema</i>
-----------------	-----------------------------------

Description

Unbind the JSON schema bound to the entity.

Usage

```
synUnbindSchema(instance, synapse_client=NULL)
```

Arguments

- instance (EntityView) The EntityView instance to operate on.
- synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Using this function
# Unbinding a JSON schema from an EntityView. This example demonstrates
# how to remove a schema binding from an entity. Assumes the entity
# already has a schema bound. Set the PROJECT_NAME variable to your
# project name.

library(synapser)
synLogin()
```

```

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VIEW_NAME = "test_script_entityview"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test EntityView of the files in the project
test_view = EntityView(
  name = VIEW_NAME,
  parent_id = PROJECT_ID,
  scope_ids = list(PROJECT_ID),
  view_type_mask = 1L # 1 = files
) |> synStore()
cat("Created test EntityView:", VIEW_NAME, "\n")

# Bind JSON schema to the EntityView first
bound_schema = test_view |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to EntityView:\n")
bound_schema

# Unbind the schema from the EntityView
test_view |> synUnbindSchema()
cat("Successfully unbound schema from EntityView\n")

## End(Not run)

```

Description

Unbind the JSON schema bound to the entity.

Usage

```
synUnbindSchema(instance, synapse_client=NULL)
```

Arguments

`instance` (File) The File instance to operate on.
`synapse_client` (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Using this function
# Unbinding a JSON schema from a folder or file. This example demonstrates
# how to remove schema bindings from entities. Assumes entities already have
# schemas bound. Set the PROJECT_NAME and FILE_PATH variables to your
# project name and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
```

```

)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder first
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()
cat("Created test file:", FILE_PATH, "\n")

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Unbind the schema from the folder
test_folder |> synUnbindSchema()
cat("Successfully unbound schema from folder\n")

# Unbind the schema from the file
example_file |> synUnbindSchema()
cat("Successfully unbound schema from file\n")

## End(Not run)

```

synUnbindSchema

Folder : unbind_schema

Description

Unbind the JSON schema bound to the entity.

Usage

```
synUnbindSchema(instance, synapse_client=NULL)
```

Arguments

`instance` (Folder) The Folder instance to operate on.

`synapse_client` (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Using this function
# Unbinding a JSON schema from a folder or file. This example demonstrates
# how to remove schema bindings from entities. Assumes entities already have
# schemas bound. Set the PROJECT_NAME and FILE_PATH variables to your
# project name and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")
```

```

# Bind JSON schema to the folder first
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()
cat("Created test file:", FILE_PATH, "\n")

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Unbind the schema from the folder
test_folder |> synUnbindSchema()
cat("Successfully unbound schema from folder\n")

# Unbind the schema from the file
example_file |> synUnbindSchema()
cat("Successfully unbound schema from file\n")

## End(Not run)

```

synUnbindSchema

Project : unbind_schema

Description

Unbind the JSON schema bound to the entity.

Usage

```
synUnbindSchema(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Using this function
# Unbinding a JSON schema from a project. This example demonstrates
# how to remove a schema binding from an entity. Assumes the entity
# already has a schema bound. Set the PROJECT_NAME variable to your
# project name.
library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project first
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to project:\n")
bound_schema

# Unbind the schema from the project
test_project |> synUnbindSchema()
cat("Successfully unbound schema from project\n")

## End(Not run)
```

synUnbindSchema	<i>Table : unbind_schema</i>
-----------------	------------------------------

Description

Unbind the JSON schema bound to the entity.

Usage

```
synUnbindSchema(instance, synapse_client=NULL)
```

Arguments

instance (Table) The Table instance to operate on.

synapse_client (Optional[Synapse]) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Using this function
# Unbinding a JSON schema from a table. This example demonstrates
# how to remove a schema binding from an entity. Assumes the entity
# already has a schema bound. Set the PROJECT_NAME variable to your
# project name.
library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
TABLE_NAME = "test_script_table"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
```

```

        foo = list(type = "string"),
        bar = list(type = "integer")
    ),
    required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test table
test_table = Table(name = TABLE_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test table:", TABLE_NAME, "\n")

# Bind JSON schema to the table first
bound_schema = test_table |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to table:\n")
bound_schema

# Unbind the schema from the table
test_table |> synUnbindSchema()
cat("Successfully unbound schema from table\n")

## End(Not run)

```

synUnzipGzippedFile *WikiPage : unzip_gzipped_file*

Description

Unzip the gzipped file and return the file path to the unzipped file. If the file is a markdown file, the content will be printed.

Usage

```
synUnzipGzippedFile(file_path)
```

Arguments

file_path (str) The path to the gzipped file.

Value

The file path to the unzipped file.

Examples

```
## Not run:
## Unzip a gzipped file
  file_path = "path/to/file.md.gz"
  unzipped_file_path = synUnzipGzippedFile(file_path)

## End(Not run)
```

synUpdate

*AgentSession : update***Description**

Updates an agent session. Only updates to the access level are currently supported.

Usage

```
synUpdate(instance, synapse_client=NULL)
```

Arguments

instance (AgentSession) The AgentSession instance to operate on.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

The updated AgentSession object.

Examples

```
## Not run:
## Update the access level of an existing session.
  # Construct an AgentSession with its session ID, set the access level you
  # want, and call synUpdate().
  library(synapser)
  synLogin()

  my_session = AgentSession(id = "foo")
  my_session$access_level = "READ_YOUR_PRIVATE_DATA"
  my_session = my_session |> synUpdate()

## End(Not run)
```

synUpdateAcl

*Evaluation : update_acl***Description**

Update the access control list (ACL) for this evaluation.

You can either:

1. Provide a principal_id and access_type list to update permissions for a specific user/team
2. Provide a complete ACL named list to update all permissions at once

To remove a principal from the ACL completely, provide an empty list for access_type.

The available access types are:

- "CREATE"
- "SUBMIT"
- "READ_PRIVATE_SUBMISSION"
- "DELETE_SUBMISSION"
- "UPDATE_SUBMISSION"
- "CHANGE_PERMISSIONS"
- "READ"
- "DELETE"
- "UPDATE"

Usage

```
synUpdateAcl(instance, principal_id=NULL, access_type=NULL, acl=NULL, synapse_client=NULL)
```

Arguments

- | | |
|----------------|---|
| instance | (Evaluation) The Evaluation instance to operate on. |
| principal_id | (Optional[Union[str, int]]) The Synapse user or team ID to update permissions for. |
| access_type | (Optional[list[str]]) List of permission strings to grant to the principal. If empty, the principal will be removed from the ACL. |
| acl | (Optional[named list]) A named list containing the complete ACL data to update. You can retrieve the current ACL using synGetAcl() . If provided, principal_id and access_type are ignored. |
| synapse_client | (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used. |

Value

The updated [AccessControlList](#) response object as a named list.

Errors

- An error is raised if neither (principal_id and access_type) nor acl is provided, or if the ACL object is invalid.
- An error is also raised if the service rejects the request or an HTTP error occurs.

Examples

```
## Not run:
## Example 1: Update permissions for a specific principal (user/team)
library(synapser)
synLogin()

PROJECT_ID = "syn123456" # replace with your project ID

# Store a new evaluation in the project first
evaluation = Evaluation(
  name = "My Challenge Evaluation",
  description = "Evaluation for my data challenge",
  content_source = PROJECT_ID,
  submission_instructions_message = "Submit CSV files only",
  submission_receipt_message = "Thank you for your submission!"
) |> synStore()

# Update permissions for user with ID 12345
updated_acl = evaluation |> synUpdateAcl(
  principal_id = "12345",
  access_type = list("READ", "SUBMIT")
)
updated_acl

## Example 2: Remove a principal (user/team) from the ACL
# Using the evaluation stored in Example 1

# Remove user with ID 12345 from the ACL by providing an empty list
updated_acl = evaluation |> synUpdateAcl(
  principal_id = "12345",
  access_type = list()
)
updated_acl

## Example 3: Update the entire ACL manually
# Using the evaluation stored in Example 1

# Get the current ACL
acl = evaluation |> synGetAcl()

# Modify the ACL manually
acl$resourceAccess = c(acl$resourceAccess, list(list(
  principalId = 12345L,
  accessType = list("READ", "SUBMIT")
)))
```

```
# Update with the modified ACL
updated_acl = evaluation |> synUpdateAcl(acl = acl)
updated_acl

## End(Not run)
```

synUpdateAcl	<i>Organization : update_acl</i>
--------------	----------------------------------

Description

Updates the ACL for a principal for this organization

Usage

```
synUpdateAcl(instance, principal_id, access_type, synapse_client=NULL)
```

Arguments

instance	(Organization) The Organization instance to operate on.
principal_id	(integer) the id of the principal whose permissions are to be updated
access_type	(list[character]) List of permission types (e.g., list("READ", "CREATE", "DELETE")) see: ACCESS_TYPE
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Update the ACL for a Organization
library(synapser)
synLogin()

org = Organization(name = "dpetest")
org |> synUpdateAcl(
  principal_id = 1L,
  access_type = list("READ")
)

## End(Not run)
```

synUpdateRows	<i>Dataset : update_rows</i>
---------------	------------------------------

Description

Update the values of rows in the dataset. This method can only be used to update values in custom columns. Default columns cannot be updated, but may be used as primary keys.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The primary_keys argument must contain at least one column.
- The primary_keys argument cannot contain columns that are a LIST type.
- The primary_keys argument cannot contain columns that are a JSON type.
- The values used as the primary_keys must be unique in the table. If there are multiple rows with the same values in the primary_keys the behavior is that an error will be raised.
- The columns used in primary_keys cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

Usage

synUpdateRows(instance, values, primary_keys, dry_run=False, rows_per_query=50000L, update_size_bytes

Arguments

instance	(Dataset) The Dataset instance to operate on.
values	(object) Supports storing data from the following sources: <ul style="list-style-type: none">- A string holding the path to a CSV file. The data will be read into a Pandas DataFrame. The code makes assumptions about the format of the columns in the CSV as detailed in the csv_to_pandas_df function. You may pass in additional arguments to the csv_to_pandas_df function by passing them in as extra named arguments to this function.- A named list where the key is the column name and the value is one or more values. The values will be wrapped into a Pandas DataFrame. You may pass in additional arguments to the pandas DataFrame constructor by passing them in as extra named arguments to this function. Read about the available arguments in the Pandas DataFrame documentation.- A Pandas DataFrame
primary_keys	(list[character]) The columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist nothing will be done.
dry_run	(logical) If set to TRUE the data will not be updated in Synapse. A message will be printed to the console with the number of rows that would have been updated and inserted.

rows_per_query	(integer) The number of rows that will be queried from Synapse per request. Since we need to query for the data that is being updated this will determine the number of rows that are queried at a time. The default is 50,000 rows.
update_size_bytes	(integer) The maximum size of the request that will be sent to Synapse when updating rows of data. The default is 1.9MB.
insert_size_bytes	(integer) The maximum size of the request that will be sent to Synapse when inserting rows of data. The default is 900MB.
job_timeout	(integer) The maximum amount of time to wait for a job to complete. This is used when inserting, and updating rows of data. Each individual request to Synapse will be sent as an independent job. If the timeout is reached a timeout error will be raised. The default is 600 seconds
date_columns	(Optional[list[character]]) (CSV file only) The names of columns in your CSV file that contain dates or datetimes stored as formatted strings (e.g. "2024-01-15" or "01/15/2024 13:30"). The columns are parsed with <code>pandas.to_datetime</code> and converted to epoch time in milliseconds before the data is uploaded, which is the format Synapse requires for DATE columns.
date_format	(Optional[character or named list]) (CSV file only) How the strings in <code>date_columns</code> are formatted — a strftime format string (e.g. "%m/%d/%Y") applied to every column, or a named list mapping column names to their formats. Supply this so that ambiguous dates (e.g. "01/02/2024") are not silently misinterpreted and to optimize the data upload performance. If the values in a column do not match the format an error is raised.
wait_for_eventually_consistent_view	(logical) Only used if the table is a view. If set to TRUE this will wait for the view to reflect any changes that you've made to the view. This is useful if you need to query the view after making changes to the data.
wait_for_eventually_consistent_view_timeout	(integer) The maximum amount of time to wait for a view to be eventually consistent. The default is 600 seconds.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Examples

```
## Not run:
## Update custom column values in a dataset.
library(synapser)
synLogin()

my_dataset = Dataset(id = "syn1234") |> synGet()

# my_annotation must already exist in the dataset as a custom column
```

```

modified_data = data.frame(
  id = c("syn1234"),
  my_annotation = c("good data")
)
my_dataset |> synUpdateRows(values = modified_data, primary_keys = list("id"), dry_run = FALSE)
my_dataset |> synStore()

## End(Not run)

```

synUpdateRows

DatasetCollection : update_rows

Description

Update the values of rows in the dataset collection. This method can only be used to update values in custom columns. Default columns cannot be updated, but may be used as primary keys.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an error will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

Usage

```
synUpdateRows(instance, values, primary_keys, dry_run=FALSE, rows_per_query=50000L, update_size_bytes)
```

Arguments

`instance` (DatasetCollection) The DatasetCollection instance to operate on.

`values` (object) Supports storing data from the following sources:

- A string holding the path to a CSV file. The data will be read into a **Pandas DataFrame**. The code makes assumptions about the format of the columns in the CSV as detailed in the `csv_to_pandas_df` function.
- A named list where the key is the column name and the value is one or more values. The values will be wrapped into a **Pandas DataFrame**. You may pass in additional arguments to the pandas DataFrame constructor by passing them in as keyword arguments to this function. Read about the available arguments in the **Pandas DataFrame** documentation.
- A **Pandas DataFrame**

primary_keys	(list[character]) The columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist nothing will be done.
dry_run	(logical) If set to TRUE the data will not be updated in Synapse. A message will be printed to the console with the number of rows that would have been updated and inserted. If you would like to see the data that would be updated and inserted you may set the dry_run argument to TRUE.
rows_per_query	(integer) The number of rows that will be queried from Synapse per request. Since we need to query for the data that is being updated this will determine the number of rows that are queried at a time. The default is 50,000 rows.
update_size_bytes	(integer) The maximum size of the request that will be sent to Synapse when updating rows of data. The default is 1.9MB.
insert_size_bytes	(integer) The maximum size of the request that will be sent to Synapse when inserting rows of data. The default is 900MB.
job_timeout	(integer) The maximum amount of time to wait for a job to complete. This is used when inserting, and updating rows of data. Each individual request to Synapse will be sent as an independent job. If the timeout is reached a timeout error will be raised. The default is 600 seconds
date_columns	(Optional[list[character]]) (CSV file only) The names of columns in your CSV file that contain dates or datetimes stored as formatted strings (e.g. "2024-01-15" or "01/15/2024 13:30"). The columns are parsed with pandas.to_datetime and converted to epoch time in milliseconds before the data is uploaded, which is the format Synapse requires for DATE columns.
date_format	(Optional[character or named list]) (CSV file only) How the strings in date_columns are formatted — a strftime format string (e.g. "%m/%d/%Y") applied to every column, or a named list mapping column names to their formats. Supply this so that ambiguous dates (e.g. "01/02/2024") are not silently misinterpreted and to optimize the data upload performance. If the values in a column do not match the format an error is raised.
wait_for_eventually_consistent_view	(logical) Only used if the table is a view. If set to TRUE this will wait for the view to reflect any changes that you've made to the view. This is useful if you need to query the view after making changes to the data.
wait_for_eventually_consistent_view_timeout	(integer) The maximum amount of time to wait for a view to be eventually consistent. The default is 600 seconds.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Note

When values is a path to a CSV file, you may pass additional keyword arguments to the `csv_to_pandas_df` function by adding them as extra named arguments to `synUpdateRows()`, e.g. `separator = ";"` or `lines_to_skip = 1L`.

Examples

```
## Not run:
## Update custom column values in a Dataset Collection.
library(synpser)
synLogin()

my_collection = DatasetCollection(id = "syn1234") |> synGet()

# my_annotation must already exist in the dataset collection as a custom column
modified_data = data.frame(
  id = c("syn1234"),
  my_annotation = c("good data")
)
my_collection |> synUpdateRows(values = modified_data, primary_keys = list("id"), dry_run = FALSE)
my_collection |> synStore()

## End(Not run)
```

synUpdateRows

EntityView : update_rows

Description

This method leverages the logic provided by `synUpsertRows()` to provide an interface for updating rows in a View-like entity. Update functionality will only work for values in custom columns within a View-like entity.

Limitations:

- When updating many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to update.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The `primary_keys` columns must be present in the data being updated. An error is raised if a primary key column is missing.
- The values in the `primary_keys` columns cannot be null. A null value cannot be used to match an existing row, so an error is raised if any row has a null value in a primary key column.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an error will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

Usage

```
synUpdateRows(instance, values, primary_keys, dry_run=False, rows_per_query=50000L, update_size_bytes
```

Arguments

instance	(EntityView) The EntityView instance to operate on.
values	(Union[str, dict[str, Any], object]) Supports storing data from the following sources: <ul style="list-style-type: none"> - A string holding the path to a CSV file. The data will be read into a data frame. The code makes assumptions about the format of the columns in the CSV as detailed in the csv_to_pandas_df function. - A named list where the key is the column name and the value is one or more values. The values will be wrapped into a data frame. - A data frame
primary_keys	(list[str]) The columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist nothing will be done.
dry_run	(bool) If set to TRUE the data will not be updated in Synapse. A message will be printed to the console with the number of rows that would have been updated and inserted. If you would like to see the data that would be updated and inserted you may set the dry_run argument to TRUE.
rows_per_query	(int) The number of rows that will be queried from Synapse per request. Since we need to query for the data that is being updated this will determine the number of rows that are queried at a time. The default is 50,000 rows.
update_size_bytes	(int) The maximum size of the request that will be sent to Synapse when updating rows of data. The default is 1.9MB.
insert_size_bytes	(int) The maximum size of the request that will be sent to Synapse when inserting rows of data. The default is 900MB.
job_timeout	(int) The maximum amount of time to wait for a job to complete. This is used when inserting, and updating rows of data. Each individual request to Synapse will be sent as an independent job. If the timeout is reached a timeout error will be raised. The default is 600 seconds.
date_columns	(Optional[list[character]]) (CSV file only) The names of columns in your CSV file that contain dates or datetimes stored as formatted strings (e.g. "2024-01-15" or "01/15/2024 13:30"). The columns are parsed with <code>pandas.to_datetime</code> and converted to epoch time in milliseconds before the data is uploaded, which is the format Synapse requires for DATE columns.
date_format	(Optional[character or named list]) (CSV file only) How the strings in date_columns are formatted — a strftime format string (e.g. "%m/%d/%Y") applied to every column, or a named list mapping column names to their formats. Supply this so that ambiguous dates (e.g. "01/02/2024") are not silently misinterpreted and to optimize the data upload performance. If the values in a column do not match the format an error is raised.

`wait_for_eventually_consistent_view`
 (bool) Only used if the table is a view. If set to TRUE this will wait for the view to reflect any changes that you've made to the view. This is useful if you need to query the view after making changes to the data. If you set this value to TRUE your view must contain the default id column which is the Synapse ID of the row. If you do not have this column in your view you will need to add it to the view before you can use this feature. The default is FALSE.

`wait_for_eventually_consistent_view_timeout`
 (int) The maximum amount of time to wait for a view to be eventually consistent. The default is 600 seconds.

`synapse_client` (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Note

When values is a path to a CSV file, you may pass additional keyword arguments to the `csv_to_pandas_df` function by adding them as extra named arguments to `synUpdateRows()`, e.g. `separator = ";"` or `lines_to_skip = 1L`.

Examples

```
## Not run:
## Updating custom annotation columns in a view
# Rows are matched on the `id` column, and the values in `species` are updated.
library(synapser)
synLogin()

my_view = EntityView(id = "syn123") |> synGet()

modified_data = data.frame(
  id = c("syn456", "syn789"),
  species = c("Homo sapiens", "Mus musculus")
)
my_view |> synUpdateRows(values = modified_data, primary_keys = list("id"), dry_run = FALSE)
my_view |> synStore()

## End(Not run)
```

synUpsertRows

Table : *upsert_rows*

Description

This method allows you to perform an upsert (Update and Insert) for row(s). This means that you may update a row with only the data that you want to change. When supplied with a row that does not match the given `primary_keys` a new row will be inserted.

Using the `primary_keys` argument you may specify which columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist it will be inserted.

Limitations:

- The request to update, and the request to insert data does not occur in a single transaction. This means that the update of data may succeed, but the insert of data may fail. Additionally, as noted in the limitation below, if data is chunked up into multiple requests you may find that a portion of your data is updated, but another portion is not.
- The number of rows that may be upserted in a single call should be kept to a minimum (< 50,000). There is significant overhead in the request to Synapse for each row that is upserted. If you are upserting a large number of rows a better approach may be to query for the data you want to update, update the data, then use the `synStoreRows()` function to update the data in Synapse. Any rows you want to insert may be added to the data frame that is passed to the `synStoreRows()` function.
- When upserting many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to upsert, or note the above limitation.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The `primary_keys` columns must be present in the data being upserted. An error is raised if a primary key column is missing.
- The values in the `primary_keys` columns cannot be null. A null value cannot be used to match an existing row, so an error is raised if any row has a null value in a primary key column.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an exception will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

The following is a Sequence Diagram that describes the upsert process at a high level:

```
sequenceDiagram
    participant User
    participant Table
    participant Synapse

    User->>Table: upsert_rows()

    loop Query and Process Updates in Chunks (rows_per_query)
        Table->>Synapse: Query existing rows using primary keys
        Synapse-->>Table: Return matching rows
        Note Over Table: Create partial row updates
    end

    loop For results from query
```

```

    Note Over Table: Sum row/chunk size
    alt Chunk size exceeds update_size_bytes
      Table->>Synapse: Push update chunk
      Synapse-->>Table: Acknowledge update
    end
    Table->>Table: Add row to chunk
  end

  alt Remaining updates exist
    Table->>Synapse: Push final update chunk
    Synapse-->>Table: Acknowledge update
  end
end

alt New rows exist
  Table->>Table: Identify new rows for insertion
  Table->>Table: Call synStoreRows() function
end

Table-->>User: Upsert complete

```

Usage

```
synUpsertRows(instance, values, primary_keys, dry_run=False, rows_per_query=50000L, update_size_bytes
```

Arguments

instance	(Table) The Table instance to operate on.
values	(Union[str, dict[str, Any], object]) Supports storing data from the following sources: <ul style="list-style-type: none"> - A string holding the path to a CSV file. The data will be read into a Pandas DataFrame. The code makes assumptions about the format of the columns in the CSV as detailed in the csv_to_pandas_df function. - A named list where the key is the column name and the value is one or more values. The values will be wrapped into a Pandas DataFrame. You may pass in additional arguments to the pandas DataFrame constructor by passing them in as keyword arguments to this function. Read about the available arguments in the Pandas DataFrame documentation. - A Pandas DataFrame
primary_keys	(list[str]) The columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist it will be inserted.
dry_run	(bool) If set to TRUE the data will not be updated in Synapse. A message will be printed to the console with the number of rows that would have been updated and inserted. If you would like to see the data that would be updated and inserted you may set the dry_run argument to TRUE.

rows_per_query	(int) The number of rows that will be queried from Synapse per request. Since we need to query for the data that is being updated this will determine the number of rows that are queried at a time. The default is 50,000 rows.
update_size_bytes	(int) The maximum size of the request that will be sent to Synapse when updating rows of data. The default is 1.9MB.
insert_size_bytes	(int) The maximum size of the request that will be sent to Synapse when inserting rows of data. The default is 900MB.
job_timeout	(int) The maximum amount of time to wait for a job to complete. This is used when inserting, and updating rows of data. Each individual request to Synapse will be sent as an independent job. If the timeout is reached a timeout error will be raised. The default is 600 seconds
date_columns	(Optional[list[character]]) (CSV file only) The names of columns in your CSV file that contain dates or datetimes stored as formatted strings (e.g. "2024-01-15" or "01/15/2024 13:30"). The columns are parsed with <code>pandas.to_datetime</code> and converted to epoch time in milliseconds before the data is uploaded, which is the format Synapse requires for DATE columns.
date_format	(Optional[character or named list]) (CSV file only) How the strings in <code>date_columns</code> are formatted — a strftime format string (e.g. "%m/%d/%Y") applied to every column, or a named list mapping column names to their formats. Supply this so that ambiguous dates (e.g. "01/02/2024") are not silently misinterpreted and to optimize the data upload performance. If the values in a column do not match the format an error is raised.
synapse_client	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

NULL

Note

When `values` is a path to a CSV file, you may pass additional keyword arguments to the `csv_to_pandas_df` function by adding them as extra named arguments to `synUpsertRows()`, e.g. `separator = ";"` or `lines_to_skip = 1L`.

Examples

```
## Not run:
## Example 1: Updating 2 rows and inserting 1 row
# In this given example we have a table with the following data:
```

#	col1	col2	col3
#	-----	-----	-----
#	A	1	1
#	B	2	2

```

# The following code will update the first row's col2 to 22, update the
# second row's col3 to 33, and insert a new row:
library(synapser)
synLogin()

# Also works with Dataset
table = Table(id = "syn123") |> synGet(table_options = TableOptions(include_columns = TRUE))

df = data.frame(
  col1 = c("A", "B", "C"),
  col2 = c(22, 2, 3),
  col3 = c(1, 33, 3)
)

table |> synUpsertRows(values = df, primary_keys = list("col1"))

# The resulting table will look like this:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 22   | 1    |
# | B    | 2    | 33   |
# | C    | 3    | 3    |

## Example 2: Deleting data from a specific cell
# In this given example we have a table with the following data:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    | 1    | 1    |
# | B    | 2    | 2    |

# The following code will clear the value in the first row's col2, and the
# second row's col3:
library(synapser)
synLogin()

# Also works with Dataset
table = Table(id = "syn123") |> synGet(table_options = TableOptions(include_columns = TRUE))

data_to_upsert = data.frame(
  col1 = c("A", "B"),
  col2 = c(NA, 2),
  col3 = c(1, NA)
)

table |> synUpsertRows(values = data_to_upsert, primary_keys = list("col1"))

# The resulting table will look like this:

# | col1 | col2 | col3 |
# |-----|-----| ----|
# | A    |      | 1    |

```

```
# | B      | 2      |      |

## Example 3: Upserting data with date columns
# In this given example we have a table with the following data:

# | col1 | date_col |
# |-----|-----|
# | A    | 2024-01-15 |
# | B    | 2024-02-20 |

# Suppose we have a CSV file with the following data that we want to upsert:

# | col1 | date_col |
# |-----|-----|
# | A    | 03/10/2024 |
# | C    | 04/01/2024 |

# The date_columns/date_format arguments tell the client how to parse
# the formatted date strings in the CSV file so that they can be compared
# against, and stored in, the table's DATE column. The following code
# will update row A's date_col and insert a new row for C:
library(synapser)
synLogin()

# Also works with Dataset
Table(id = "syn123") |> synUpsertRows(
  values = "path/to/file.csv",
  primary_keys = list("col1"),
  date_columns = list("date_col"),
  date_format = "%m/%d/%Y"
)

# The resulting table will look like this:

# | col1 | date_col |
# |-----|-----|
# | A    | 2024-03-10 |
# | B    | 2024-02-20 |
# | C    | 2024-04-01 |

## End(Not run)
```

synValidateSchema	EntityView : validate_schema
-------------------	------------------------------

Description

Validate the entity against the bound JSON schema.

Usage

```
synValidateSchema(instance, synapse_client=NULL)
```

Arguments

instance (EntityView) The EntityView instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation results.

Examples

```
## Not run:
## Using this function
# Validating an EntityView against the bound JSON schema. This example demonstrates
# how to validate an entity with annotations against its bound schema. Requires
# the entity to have a schema already bound. Set the PROJECT_NAME variable to
# your project name.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VIEW_NAME = "test_script_entityview"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)
```

```

# Create a test EntityView of the files in the project
test_view = EntityView(
    name = VIEW_NAME,
    parent_id = PROJECT_ID,
    scope_ids = list(PROJECT_ID),
    view_type_mask = 1L # 1 = files
) |> synStore()
cat("Created test EntityView:", VIEW_NAME, "\n")

# Bind JSON schema to the EntityView
bound_schema = test_view |> synBindSchema(
    json_schema_uri = SCHEMA_URI,
    enable_derived_annotations = TRUE
)
cat("Bound schema to EntityView:\n")
bound_schema

# Validate the EntityView against the bound schema
test_view = test_view |> synSetProperties(
    annotations = list(foo = "test_value", bar = 42) # Example annotations
) |> synStore()
cat("Added annotations to EntityView and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response = test_view |> synValidateSchema()
cat("EntityView validation response:\n")
validation_response

## End(Not run)

```

synValidateSchema	<i>File : validate_schema</i>
-------------------	-------------------------------

Description

Validate the entity against the bound JSON schema.

Usage

```
synValidateSchema(instance, synapse_client=NULL)
```

Arguments

instance	(File) The File instance to operate on.
synapse_client	(Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation results.

Examples

```
## Not run:
## Using this function
# Validating a folder or file against the bound JSON schema. This example demonstrates
# how to validate entities with annotations against their bound schemas. Requires entities
# to have schemas already bound. Set the PROJECT_NAME and FILE_PATH variables to your project name
# and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()
```

```

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Validate the folder entity against the bound schema
test_folder = test_folder |> synSetProperties(
  annotations = list(foo = "test_value", bar = 42)
) |> synStore()
cat("Added annotations to folder and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response = test_folder |> synValidateSchema()
cat("Folder validation response:\n")
validation_response

# Validate the file entity against the bound schema
example_file = example_file |> synSetProperties(
  annotations = list(foo = "test_value", bar = 43)
) |> synStore()
cat("Added annotations to file and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response_file = example_file |> synValidateSchema()
cat("File validation response:\n")
validation_response_file

## End(Not run)

```

synValidateSchema	<i>Folder : validate_schema</i>
-------------------	---------------------------------

Description

Validate the entity against the bound JSON schema.

Usage

```
synValidateSchema(instance, synapse_client=NULL)
```

Arguments

instance	(Folder) The Folder instance to operate on.
synapse_client	(Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation results.

Examples

```
## Not run:
## Using this function
# Validating a folder or file against the bound JSON schema. This example demonstrates
# how to validate entities with annotations against their bound schemas. Requires entities
# to have schemas already bound. Set the PROJECT_NAME and FILE_PATH variables to your project name
# and file path respectively.

library(synapser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name
FILE_PATH = "~/Sample.txt" # replace with your test file path

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
FOLDER_NAME = "test_script_folder"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test folder
test_folder = Folder(name = FOLDER_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test folder:", FOLDER_NAME, "\n")

# Bind JSON schema to the folder
bound_schema = test_folder |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to folder:\n")
```



```
bound_schema

# Create and bind schema to a file
example_file = File(path = FILE_PATH, parent_id = test_folder$id) |> synStore()

bound_schema_file = example_file |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to file:\n")
bound_schema_file

# Validate the folder entity against the bound schema
test_folder = test_folder |> synSetProperties(
  annotations = list(foo = "test_value", bar = 42)
) |> synStore()
cat("Added annotations to folder and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response = test_folder |> synValidateSchema()
cat("Folder validation response:\n")
validation_response

# Validate the file entity against the bound schema
example_file = example_file |> synSetProperties(
  annotations = list(foo = "test_value", bar = 43)
) |> synStore()
cat("Added annotations to file and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response_file = example_file |> synValidateSchema()
cat("File validation response:\n")
validation_response_file

## End(Not run)
```

synValidateSchema	<i>Project : validate_schema</i>
-------------------	----------------------------------

Description

Validate the entity against the bound JSON schema.

Usage

```
synValidateSchema(instance, synapse_client=NULL)
```

Arguments

instance (Project) The Project instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation results.

Examples

```
## Not run:
## Using this function
# Validating a project against the bound JSON schema. This example demonstrates
# how to validate an entity with annotations against its bound schema. Requires
# the entity to have a schema already bound. Set the PROJECT_NAME variable to
# your project name.
library(synpser)
synLogin()

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Get the test project
test_project = Project(id = PROJECT_ID)

# Bind JSON schema to the project
bound_schema = test_project |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
```

```

        enable_derived_annotations = TRUE
    )
    cat("Bound schema to project:\n")
    bound_schema

    # Validate the project entity against the bound schema
    test_project = test_project |> synSetProperties(
        annotations = list(foo = "test_value", bar = 42L)
    ) |> synStore()
    cat("Added annotations to project and stored\n")
    Sys.sleep(2) # Allow time for processing

    validation_response = test_project |> synValidateSchema()
    cat("Project validation response:\n")
    validation_response

## End(Not run)

```

synValidateSchema	<i>Table : validate_schema</i>
-------------------	--------------------------------

Description

Validate the entity against the bound JSON schema.

Usage

```
synValidateSchema(instance, synapse_client=NULL)
```

Arguments

instance (Table) The Table instance to operate on.

synapse_client (Optional[Synapse], optional) The Synapse client instance. If not provided, the last created instance from the Synapse class constructor will be used.

Value

The validation results.

Examples

```

## Not run:
## Using this function
# Validating a table against the bound JSON schema. This example demonstrates
# how to validate an entity with annotations against its bound schema. Requires
# the entity to have a schema already bound. Set the PROJECT_NAME variable to
# your project name.
library(synapser)
synLogin()

```

```

# Define Project and JSON schema info
PROJECT_NAME = "test_json_schema_project" # replace with your project name

PROJECT_ID = synFindEntityId(name = PROJECT_NAME)
ORG_NAME = "UniqueOrg" # replace with your organization name
SCHEMA_NAME = "myTestSchema" # replace with your schema name
TABLE_NAME = "test_script_table"
VERSION = "0.0.1"
SCHEMA_URI = sprintf("%s-%s-%s", ORG_NAME, SCHEMA_NAME, VERSION)

# Create the organization and JSON schema used in this example.
my_test_org = SchemaOrganization(name = ORG_NAME) |> synStore()

schema_definition = list(
  "$id" = "mySchema",
  type = "object",
  properties = list(
    foo = list(type = "string"),
    bar = list(type = "integer")
  ),
  required = list("foo")
)
test_schema = JSONSchema(name = SCHEMA_NAME, organization_name = ORG_NAME) |> synStore(
  json_schema_options = StoreJSONSchemaOptions(schema_body = schema_definition, version = VERSION)
)

# Create a test table
test_table = Table(name = TABLE_NAME, parent_id = PROJECT_ID) |> synStore()
cat("Created test table:", TABLE_NAME, "\n")

# Bind JSON schema to the table
bound_schema = test_table |> synBindSchema(
  json_schema_uri = SCHEMA_URI,
  enable_derived_annotations = TRUE
)
cat("Bound schema to table:\n")
bound_schema

# Validate the table entity against the bound schema
test_table = test_table |> synSetProperties(
  annotations = list(foo = "test_value", bar = 42)
) |> synStore()
cat("Added annotations to table and stored\n")
Sys.sleep(2) # Allow time for processing

validation_response = test_table |> synValidateSchema()
cat("Table validation response:\n")
validation_response

## End(Not run)

```

synWalk	<i>Folder : walk</i>
---------	----------------------

Description

Traverse through the hierarchy of entities stored under this container. Mimics `os.walk()` behavior but yields `EntityHeader` objects, with optional ASCII tree display that continues printing as the walk progresses.

Usage

```
synWalk(instance, follow_link=False, include_types=NULL, recursive=True, display_ascii_tree=False, synapse_client=None)
```

Arguments

<code>instance</code>	(Folder) The Folder instance to operate on.
<code>follow_link</code>	(bool) Whether to follow a link entity or not. Links can be used to point at other Synapse entities.
<code>include_types</code>	(Optional[list[str]]) Must be a list of entity types (ie. <code>list("folder", "file")</code>) which can be found here . Defaults to <code>list("folder", "file", "table", "entityview", "dockerrepo", "submissionview", "dataset", "datasetcollection", "materializedview", "virtualtable", "searchindex")</code> . The "folder" type is always included so the hierarchy can be traversed.
<code>recursive</code>	(bool) Whether to recursively traverse subdirectories. Defaults to <code>TRUE</code> .
<code>display_ascii_tree</code>	(bool) If <code>TRUE</code> , display an ASCII tree representation as the container structure is traversed. Tree lines are printed incrementally as each container is visited. Defaults to <code>FALSE</code> .
<code>synapse_client</code>	(Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A `GeneratorWrapper` object that yields a list of (`dirpath`, `dirs`, `nondirs`) where:

- `dirpath`: A vector of (`directory_name`, `synapse_id`) representing the current directory
- `dirs`: A list of `EntityHeader` objects for subdirectories (folders)
- `nondirs`: A list of `EntityHeader` objects for non-directory entities (files, tables, etc.)

Use `as.list()` to materialize all results into a list, or `nextElem()` to retrieve one result at a time.

Note

Each `EntityHeader` contains complete metadata including `id`, `name`, `type`, `creation/modification dates`, `version information`, and other Synapse properties. The directory path is built using `os.path.join()` to create hierarchical paths, like:

```
my-container-name/bulk-upload/file_script_folder/file_script_sub_folder
```

When `display_ascii_tree=TRUE`, the ASCII tree structure is displayed in proper hierarchical order as the entire structure is traversed sequentially. The tree will be printed in the correct parent-child relationships, but results will still be yielded as expected during the traversal process. The tree is printed progressively as each container is visited during the walk, making it suitable for very large and deeply nested structures.

Examples

```
## Not run:
## Example 1: Traverse all entities in a container
# Basic usage - traverse all entities in a container
library(synapser)
synLogin()

container = Folder(id = "syn12345")
for (entry in as.list(container |> synWalk())) {
  dirpath = entry[[1]]
  dirs = entry[[2]]
  nondirs = entry[[3]]
  cat(sprintf("Directory: %s (%s)\n", dirpath[[1]], dirpath[[2]]))

  # Print folders
  for (folder_entity in dirs) {
    cat(sprintf("  Folder: %s\n", folder_entity))
  }

  # Print files and other entities
  for (entity in nondirs) {
    cat(sprintf("  File: %s\n", entity))
  }
}

## Example 2: Display progressive ASCII tree as walk proceeds
library(synapser)
synLogin()

container = Folder(id = "syn12345")
# Display tree structure progressively as walk proceeds
for (entry in as.list(container |> synWalk(display_ascii_tree = TRUE))) {
  dirpath = entry[[1]]
  nondirs = entry[[3]]
  # Process each directory as usual
  cat(sprintf("Processing: %s\n", dirpath[[1]]))
  for (file_entity in nondirs) {
    cat(sprintf("  Found file: %s\n", file_entity$name))
  }
}

# Example output:
#
# === Container Structure ===
# my-container-name (syn52948289) [Project]
```

```

# |-- bulk-upload (syn68884548) [Folder]
# |   |-- file_script_folder (syn68884547) [Folder]
# |       |-- file_script_sub_folder (syn68884549) [Folder]
# |           |-- file_in_a_sub_folder.txt (syn68884556) [File]
# |-- root (syn67590143) [Folder]
# |   |-- subdir1 (syn67590144) [Folder]
# |       |-- file1.txt (syn67590261) [File]
# |       |-- file2.txt (syn67590287) [File]
# |-- temp-files (syn68884954) [Folder]
#     |-- root (syn68884955) [Folder]
#         |-- subdir1 (syn68884956) [Folder]
#             |-- file1.txt (syn68884959) [File]
#             |-- file2.txt (syn68884999) [File]
#
# Directory: My uniquely named project about Alzheimer's Disease (syn53185532)
#   File: EntityHeader(name = "Gene Expression Data", id = "syn66227753",
#     type = "org.sagebionetworks.repo.model.table.TableEntity", version_number = 1,
#     version_label = "in progress", is_latest_version = TRUE, benefactor_id = 53185532,
#     created_on = "2025-04-11T21:24:28.913Z", modified_on = "2025-04-11T21:24:34.996Z",
#     created_by = "3481671", modified_by = "3481671")
#   Folder: EntityHeader(name = "Research Data", id = "syn68327923",
#     type = "org.sagebionetworks.repo.model.Folder", version_number = 1,
#     version_label = "1", is_latest_version = TRUE, benefactor_id = 68327923,
#     created_on = "2025-06-16T21:51:50.460Z", modified_on = "2025-06-16T22:19:41.481Z",
#     created_by = "3481671", modified_by = "3481671")

```

```
## End(Not run)
```

synWalk

Project : walk

Description

Traverse through the hierarchy of entities stored under this container. Mimics `os.walk()` behavior but yields `EntityHeader` objects, with optional ASCII tree display that continues printing as the walk progresses.

Usage

```
synWalk(instance, follow_link=FALSE, include_types=NULL, recursive=TRUE, display_ascii_tree=FALSE, sy
```

Arguments

<code>instance</code>	(Project) The Project instance to operate on.
<code>follow_link</code>	(bool) Whether to follow a link entity or not. Links can be used to point at other Synapse entities.
<code>include_types</code>	(Optional[list[str]]) Must be a list of entity types (ie. <code>list("folder", "file")</code>) which can be found here . Defaults to <code>list("folder", "file", "table", "entityview", "dockerrepo", "submissionview", "dataset", "datasetcollection", "materializedview",</code>

"virtualtable", "searchindex"). The "folder" type is always included so the hierarchy can be traversed.

recursive (bool) Whether to recursively traverse subdirectories. Defaults to TRUE.

display_ascii_tree (bool) If TRUE, display an ASCII tree representation as the container structure is traversed. Tree lines are printed incrementally as each container is visited. Defaults to FALSE.

synapse_client (Optional[Synapse]) If not passed in, the last created instance from the Synapse class constructor will be used.

Value

A list of (dirpath, dirs, nondirs) where:

- dirpath: A vector of (directory_name, synapse_id) representing the current directory
- dirs: A list of EntityHeader objects for subdirectories (folders)
- nondirs: A list of EntityHeader objects for non-directory entities (files, tables, etc.)

Note

Each EntityHeader contains complete metadata including id, name, type, creation/modification dates, version information, and other Synapse properties. The directory path is built using `os.path.join()` to create hierarchical paths, like "my-container-name/bulk-upload/file_script_folder/file_script_sub_folder".

When `display_ascii_tree=TRUE`, the ASCII tree structure is displayed in proper hierarchical order as the entire structure is traversed sequentially. The tree will be printed in the correct parent-child relationships, but results will still be yielded as expected during the traversal process. When `display_ascii_tree=TRUE`, the tree is printed progressively as each container is visited during the walk, making it suitable for very large and deeply nested structures.

Examples

```
## Not run:
## Example 1: Traverse all entities in a container
# Basic usage - traverse all entities in a container
library(synapser)
synLogin()

container = Project(id = "syn12345")
for (item in as.list(container |> synWalk())) {
  dirpath = item[[1]]
  dirs = item[[2]]
  nondirs = item[[3]]

  cat(sprintf("Directory: %s (%s)\n", dirpath[1], dirpath[2]))

  # Print folders
  for (folder_entity in dirs) {
    cat(sprintf("  Folder: %s\n", folder_entity))
  }
}
```



```

    # Print files and other entities
    for (entity in nondirs) {
        cat(sprintf(" Non-directory entity: %s\n", entity))
    }
}

## Example 2: Display progressive ASCII tree as walk proceeds
library(synapser)
synLogin()

container = Project(id = "syn12345")
# Display tree structure progressively as walk proceeds
for (item in as.list(container |> synWalk(display_ascii_tree = TRUE))) {
    dirpath = item[[1]]
    nondirs = item[[3]]

    # Process each directory as usual
    cat(sprintf("Processing: %s\n", dirpath[1]))
    for (file_entity in nondirs) {
        cat(sprintf(" Found non-directory entity: %s\n", file_entity$name))
    }
}

# Example output:
#
# === Container Structure ===
# my-container-name (syn52948289) [Project]
# |-- bulk-upload (syn68884548) [Folder]
# |   |-- file_script_folder (syn68884547) [Folder]
# |       |-- file_script_sub_folder (syn68884549) [Folder]
# |           |-- file_in_a_sub_folder.txt (syn68884556) [File]
# |-- root (syn67590143) [Folder]
# |   |-- subdir1 (syn67590144) [Folder]
# |       |-- file1.txt (syn67590261) [File]
# |       |-- file2.txt (syn67590287) [File]
# |-- temp-files (syn68884954) [Folder]
#     |-- root (syn68884955) [Folder]
#         |-- subdir1 (syn68884956) [Folder]
#             |-- file1.txt (syn68884959) [File]
#             |-- file2.txt (syn68884999) [File]
#
# Directory: My uniquely named project about Alzheimer's Disease (syn53185532)
# File: EntityHeader(name='Gene Expression Data', id='syn66227753',
#   type='org.sagebionetworks.repo.model.table.TableEntity', version_number=1,
#   version_label='in progress', is_latest_version=True, benefactor_id=53185532,
#   created_on='2025-04-11T21:24:28.913Z', modified_on='2025-04-11T21:24:34.996Z',
#   created_by='3481671', modified_by='3481671')
# Folder: EntityHeader(name='Research Data', id='syn68327923',
#   type='org.sagebionetworks.repo.model.Folder', version_number=1,
#   version_label='1', is_latest_version=True, benefactor_id=68327923,
#   created_on='2025-06-16T21:51:50.460Z', modified_on='2025-06-16T22:19:41.481Z',
#   created_by='3481671', modified_by='3481671')
```

```
## End(Not run)
```

Table	<i>Table</i>
-------	--------------

Description

A Table represents the metadata of a table.

Usage

Table(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=NULL, columns=NULL)

Arguments

id	(Optional[str]) The unique immutable ID for this table. A new ID will be generated for new Tables. Once issued, this ID is guaranteed to never change or be re-issued
name	(Optional[str]) The name of this table. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses
parent_id	(Optional[str]) The ID of the Entity that is the parent of this table.
activity	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse. It is analogous to the Activity defined in the W3C Specification on Provenance. Activity cannot be removed during a store operation by setting it to None. You must use: synDelete or synDisassociateActivityFromEntity .
version_number	(Optional[int]) (Read Only) The version number issued to this version on the object. Use this synSnapshot() function to create a new version of the table.
description	(Optional[str]) The description of this entity. Must be 1000 characters or less.
columns	(Optional[Union[list[Column], OrderedDict[str, Column], dict[str, Column]]]) The columns of this table. This is an ordered named list where the key is the name of the column and the value is the Column object. When creating a new instance of a Table object you may pass any of the following types as the columns argument:

- A list of Column objects
- A named list where the key is the name of the column and the value is the Column object

The order of the columns will be the order they are stored in Synapse. If you need to reorder the columns the recommended approach is to use the [synReorderColumn\(\)](#) function. Additionally, you may add, and delete columns using the [synAddColumn\(\)](#), and [synDeleteColumn\(\)](#) functions on your Table instance.

You may modify the attributes of the Column object to change the column type,

name, or other attributes. For example suppose I'd like to change a column from a INTEGER to a DOUBLE. I can do so by changing the column type attribute of the Column object. The next time you store the table the column will be updated in Synapse with the new type.

```
library(synapser)
synLogin()
```

```
table = Table(id = "syn1234") |> synGet()
table$columns[["my_column"]]$column_type = "DOUBLE"
table = table |> synStore()
```

Note that the keys in this named list should match the column names as they are in Synapse. However, know that the name attribute of the Column object is used for all interactions with the Synapse API. The named list key is purely for the usage of this interface. For example, if you wish to rename a column you may do so by changing the name attribute of the Column object. The key in the named list does not need to be changed. The next time you store the table the column will be updated in Synapse with the new name and the key in the named list will be updated.

etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
created_on	(Optional[str]) The date this table was created.
created_by	(Optional[str]) The ID of the user that created this table.
modified_on	(Optional[str]) The date this table was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format
modified_by	(Optional[str]) The ID of the user that last modified this table.
version_label	(Optional[str]) (Read Only) The version label for this table. Use the synSnapshot() function to create a new version of the table.
version_comment	(Optional[str]) (Read Only) The version comment for this table. Use the synSnapshot() function to create a new version of the table.
is_latest_version	(Optional[bool]) (Read Only) If this is the latest version of the object.
is_search_enabled	(Optional[bool]) When creating or updating a table or view specifies if full text search should be enabled. Note that enabling full text search might slow down the indexing of the table or view.
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the table. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list <code>list()</code> or <code>NULL</code> and store the entity.

Format

An S3-style class object.

Value

NULL

Methods

- `Table(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=NULL, columns=list(), etag=NULL, created_on=NULL, created_by=NULL, modified_on=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, is_search_enabled=NULL, annotations=list())`: Constructor for [Table](#)
- `synAddColumn(instance, column, index=NULL)`: Add column(s) to the table. Note that this does not store the column(s) in Synapse. You must call the `synStore()` function on this Table instance to store the column(s) in Synapse. This is a convenience function to eliminate the need to manually add the column(s) to the `$columns` list.

This function will add an item to the `$columns` attribute of the Table instance. `$columns` is a named list where the key is the name of the column and the value is the Column object.

- `synBindSchema(instance, json_schema_uri, enable_derived_annotations=FALSE, synapse_client=NULL)`: Bind a JSON schema to the entity.
- `synDeleteColumn(instance, name)`: Mark a column for deletion. Note that this does not delete the column from Synapse. You must call the `synStore()` function on this Table instance to delete the column from Synapse. This is a convenience function to eliminate the need to manually delete the column from the named list and add it to the `$_columns_to_delete` attribute.
- `synDeletePermissions(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL)`: Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the `synSetPermissions()` function with the `access_type` set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synDeleteRows(instance, query=NULL, df=NULL, job_timeout=600L, synapse_client=NULL)`: Delete rows from a table given a query or a data frame to select rows. The query at a minimum must select the ROW_ID and ROW_VERSION columns. If you want to inspect the data that will be deleted ahead of time you may use the `synQuery()` function to get the data. The data frame must at least contain the ROW_ID and ROW_VERSION columns. And ROW_ETAG column is also required if the entity is one of the following: EntityView, Dataset, DatasetCollection, or SubmissionView. If both query and df are provided, the query will be used.
- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synGetSchema(instance, synapse_client=NULL)`: Get the JSON schema bound to the entity.
- `synGetSchemaDerivedKeys(instance, synapse_client=NULL)`: Retrieve derived JSON schema keys for the entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character=r"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a download_location in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a data frame or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API:

<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>

- `synReorderColumn(instance, name, index)`: Reorder a column in the table. Note that this does not store the column in Synapse. You must call the `synStore()` function on this Table instance to store the column in Synapse. This is a convenience function to eliminate the need

to manually reorder the named list stored in the `$columns` attribute.

You must ensure that the index is within the bounds of the number of columns in the table. If you pass in an index that is out of bounds the column will be added to the end of the list.

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.
- `synSnapshot(instance, comment=NULL, label=NULL, include_activity=TRUE, associate_activity_to_new_synapse_client=NULL)`: Request to create a new snapshot of a table. The provided comment, label, and activity will be applied to the current version thereby creating a snapshot and locking the current version. After the snapshot is created a new version will be started with an 'in-progress' label.
- `synStoreRows(instance, values, schema_storage_strategy=NULL, column_expansion_strategy=NULL, dry_run=FALSE, additional_changes=NULL, insert_size_bytes=943718400L, csv_table_descriptor=NULL, date_columns=NULL, date_format=NULL, read_csv_kwargs=NULL, to_csv_kwargs=NULL, job_timeout=600L, synapse_client=NULL)`: Add or update rows in Synapse from the sources defined below. In most cases the result of this function call will append rows to the table. In the case of an update this method works on a full row replacement. What this means is that you may not do a partial update of a row. If you want to update a row you must pass in all the data for that row, or the data for the columns not provided will be set to null.

If you'd like to update a row see the example [Updating rows in a table](#) below.

If you'd like to perform an upsert or partial update of a row you may use the `synUpsertRows()` function. See that function for more information.

Note the following behavior for the order of columns:

- If a column is added via the `synAddColumn()` function it will be added at the index you specify, or at the end of the columns list.
- If column(s) are added during the construction of your Table instance, ie. `Table(columns = list(Column(name = "foo")))`, they will be added at the beginning of the columns list.
- If you use the `synStoreRows()` function and the `schema_storage_strategy` is set to `INFER_FROM_DATA` the columns will be added at the end of the columns list.

How datetime values are interpreted:

Synapse DATE columns store an exact moment in time, represented as the number of milliseconds since 1970-01-01 00:00:00 UTC. Before upload, datetime values are converted to that representation as follows:

- Timezone-aware datetimes (POSIXct values created with a tz, e.g. as `POSIXct("2017-02-14 11:23", tz = "America/Los_Angeles")`) are converted to UTC exactly. This is the recommended way to pass datetime data: the stored value does not depend on the timezone settings of the machine performing the upload or on the date the upload is run.
- Naive datetimes (POSIXct values without a tz) are assumed to be in the local timezone of the machine **at the time of upload**. That single UTC offset is applied to every value, including values whose dates fall in a different daylight saving period, which will be stored shifted by one hour. For example, uploading 2017-02-14 11:23 (a PST date, UTC-8) from a machine currently on PDT (UTC-7) stores 2017-02-14 18:23 UTC instead of the correct 19:23 UTC. To avoid this, create your datetimes with a timezone, e.g. as `POSIXct(x, tz = "America/Los_Angeles")`.

- Plain R Date values (as opposed to datetimes) are treated as midnight of that date and converted the same way naive datetimes are: using the local timezone of the machine **at the time of upload**, with the same daylight-saving-shift risk described above.
- Midnight values deserve extra care: the one-hour daylight saving shift described above can move a naive midnight datetime — or a Date, which is treated as midnight — to 11 PM of the previous day, changing the calendar date the value displays as.

Why timezone-aware values are recommended:

A naive datetime such as 2017-02-14 11:23 is only "what a clock on the wall said." Before it can be stored as an exact moment in time, something has to answer: *a clock where?* Timezone-aware values answer that question in the data itself; naive values leave the client to guess, and it guesses the uploading machine's current timezone.

A zone name like "America/Los_Angeles" does not mean a fixed offset such as UTC-8. It means "a Los Angeles wall clock" — and those clocks move twice a year: UTC-7 in summer (PDT), UTC-8 in winter (PST). When you create datetimes with `as.POSIXct(x, tz = "America/Los_Angeles")`, R looks up what LA clocks were set to on each value's own date:

- 2017-02-14 → winter → interpreted using UTC-8
- 2018-10-01 → summer → interpreted using UTC-7

When the data is queried back with `synQuery(..., convert_to_datetime = TRUE)`, DATE columns are returned as datetimes in UTC. See `vignette("tables", package = "synapser")` for how to work with the returned values and view them in another timezone.

Limitations:

- Synapse limits the number of rows that may be stored in a single request to a CSV file that is 1GB. If you are storing a CSV file that is larger than this limit the data will be chunked into smaller requests. This process is done by reading the file once to determine what the row and byte boundaries are and calculating the MD5 hash of that portion, then reading the file again to send the data to Synapse. This process is done to ensure that the data is not corrupted during the upload process, in addition Synapse requires the MD5 hash of the data to be sent in the request along with the number of bytes that are being sent.
- The limit of 1GB is also enforced when storing a named list or a data frame. The data will be converted to a CSV format using the `to_csv()` pandas function. If you are storing more than a 1GB file it is recommended that you store the data as a CSV and use the file path to upload the data. This is due to the fact that the data frame chunking process is slower than reading portions of a file on disk and calculating the MD5 hash of that portion.

The following is a Sequence Diagram that describes the process noted in the limitation above. It shows how the data is chunked into smaller requests when the data exceeds the limit of 1GB, and how portions of the data are read from the CSV file on disk while being uploaded to Synapse.

```
sequenceDiagram
    participant User
    participant Table
    participant FileSystem
    participant Synapse

    User->>Table: store_rows(values)

    alt CSV size > 1GB
        Table->>Synapse: Apply schema changes before uploading
```

```

note over Table, FileSystem: Read CSV twice
Table->>FileSystem: Read entire CSV (First Pass)
FileSystem-->>Table: Compute chunk sizes & MD5 hashes

loop Read and Upload CSV chunks (Second Pass)
  Table->>FileSystem: Read next chunk from CSV
  FileSystem-->>Table: Return bytes
  Table->>Synapse: Upload CSV chunk
  Synapse-->>Table: Return file_handle_id
  Table->>Synapse: Send 'TableUpdateTransaction' to append/update rows
  Synapse-->>Table: Transaction result
end
else
  Table->>Synapse: Upload CSV without splitting & Any additional schema changes
  Synapse-->>Table: Return file_handle_id
  Table->>Synapse: Send 'TableUpdateTransaction' to append/update rows
  Synapse-->>Table: Transaction result
end

Table-->>User: Upload complete

```

The following is a Sequence Diagram that describes the process noted in the limitation above for data frames. It shows how the data is chunked into smaller requests when the data exceeds the limit of 1GB, and how portions of the data are read from the data frame while being uploaded to Synapse.

```

sequenceDiagram
    participant User
    participant Table
    participant MemoryBuffer
    participant Synapse

    User->>Table: store_rows(DataFrame)

    loop For all rows in DataFrame in 100 row increments
        Table->>MemoryBuffer: Convert DataFrame rows to CSV in-memory
        MemoryBuffer-->>Table: Compute chunk sizes & MD5 hashes
    end

    alt Multiple chunks detected
        Table->>Synapse: Apply schema changes before uploading
    end

    loop For all chunks found in first loop
        loop for all parts in chunk byte boundry
            Table->>MemoryBuffer: Read small (< 8MB) part of the chunk
            MemoryBuffer-->>Table: Return bytes (with correct offset)
            Table->>Synapse: Upload part

```



```

        Synapse-->>Table: Upload response
    end
    Table-->>Synapse: Complete upload
    Synapse-->>Table: Return file_handle_id
    Table-->>Synapse: Send 'TableUpdateTransaction' to append/update rows
    Synapse-->>Table: Transaction result
end

Table-->>User: Upload complete

```

- `synUnbindSchema(instance, synapse_client=NULL)`: Unbind the JSON schema bound to the entity.
- `synUpsertRows(instance, values, primary_keys, dry_run=FALSE, rows_per_query=50000L, update_size_bytes=1992294.4, insert_size_bytes=943718400L, job_timeout=600L, date_columns=NULL, date_format=NULL, synapse_client=NULL)`: This method allows you to perform an upsert (Update and Insert) for row(s). This means that you may update a row with only the data that you want to change. When supplied with a row that does not match the given `primary_keys` a new row will be inserted.

Using the `primary_keys` argument you may specify which columns to use to determine if a row already exists. If a row exists with the same values in the columns specified in this list the row will be updated. If a row does not exist it will be inserted.

Limitations:

- The request to update, and the request to insert data does not occur in a single transaction. This means that the update of data may succeed, but the insert of data may fail. Additionally, as noted in the limitation below, if data is chunked up into multiple requests you may find that a portion of your data is updated, but another portion is not.
- The number of rows that may be upserted in a single call should be kept to a minimum (< 50,000). There is significant overhead in the request to Synapse for each row that is upserted. If you are upserting a large number of rows a better approach may be to query for the data you want to update, update the data, then use the `synStoreRows()` function to update the data in Synapse. Any rows you want to insert may be added to the data frame that is passed to the `synStoreRows()` function.
- When upserting many rows the requests to Synapse will be chunked into smaller requests. The limit is 2MB per request. This chunking will happen automatically and should not be a concern for most users. If you are having issues with the request being too large you may lower the number of rows you are trying to upsert, or note the above limitation.
- The `primary_keys` argument must contain at least one column.
- The `primary_keys` argument cannot contain columns that are a LIST type.
- The `primary_keys` argument cannot contain columns that are a JSON type.
- The `primary_keys` columns must be present in the data being upserted. An error is raised if a primary key column is missing.
- The values in the `primary_keys` columns cannot be null. A null value cannot be used to match an existing row, so an error is raised if any row has a null value in a primary key column.
- The values used as the `primary_keys` must be unique in the table. If there are multiple rows with the same values in the `primary_keys` the behavior is that an exception will be raised.
- The columns used in `primary_keys` cannot contain updated values. Since the values in these columns are used to determine if a row exists, they cannot be updated in the same transaction.

The following is a Sequence Diagram that describes the upsert process at a high level:

```
sequenceDiagram
    participant User
    participant Table
    participant Synapse

    User->>Table: upsert_rows()

    loop Query and Process Updates in Chunks (rows_per_query)
        Table->>Synapse: Query existing rows using primary keys
        Synapse-->>Table: Return matching rows
        Note Over Table: Create partial row updates

        loop For results from query
            Note Over Table: Sum row/chunk size
            alt Chunk size exceeds update_size_bytes
                Table->>Synapse: Push update chunk
                Synapse-->>Table: Acknowledge update
            end
            Table->>Table: Add row to chunk
        end

        alt Remaining updates exist
            Table->>Synapse: Push final update chunk
            Synapse-->>Table: Acknowledge update
        end
    end

    alt New rows exist
        Table->>Table: Identify new rows for insertion
        Table->>Table: Call synStoreRows() function
    end

    Table-->>User: Upsert complete
```

- `synValidateSchema(instance, synapse_client=NULL)`: Validate the entity against the bound JSON schema.

Examples

```
## Not run:
## Example 1: Create a table with data without specifying columns
# This API is setup to allow the data to define which columns are created on the
# Synapse table automatically. The limitation with this behavior is that the
# columns created will only be of the following types:

# - STRING
# - LARGETEXT
# - INTEGER
```

```

# - DOUBLE
# - BOOLEAN
# - DATE

# The determination of the column type is based on the data that is passed in
# using the pandas function
# infer_dtype (https://pandas.pydata.org/docs/reference/api/pandas.api.types.infer\_dtype.html).
# If you need a more specific column type, or need to add options to the columns
# follow the examples below.
library(synapser)
synLogin()

my_data = data.frame(
  my_string_column = c("a", "b", "c", "d"),
  my_integer_column = c(1, 2, 3, 4),
  my_double_column = c(1.0, 2.0, 3.0, 4.0),
  my_boolean_column = c(TRUE, FALSE, TRUE, FALSE)
)

table = Table(
  name = "my_table",
  parent_id = "syn1234"
) |> synStore()

table |> synStoreRows(values = my_data, schema_storage_strategy = "INFER_FROM_DATA")

# Prints out the stored data about this specific column
table$columns[["my_string_column"]]

## Example 2: Rename an existing column
# This examples shows how you may retrieve a table from synapse, rename a column,
# and then store the table back in synapse.
library(synapser)
synLogin()

table = Table(
  name = "my_table",
  parent_id = "syn1234"
) |> synGet()

# You may also get the table by id:
table = Table(id = "syn4567") |> synGet()

table$columns[["my_old_column"]]$name = "my_new_column"

# Before the data is stored in synapse you'll still be able to use the old key to access the column entry
table$columns[["my_old_column"]]

table = table |> synStore()

# After the data is stored in synapse you'll be able to use the new key to access the column entry
table$columns[["my_new_column"]]

```

```

## Example 3: Create a table with a list of columns
# A list of columns may be passed in when creating a new table. The order of the
# columns in the list will be the order they are stored in Synapse. If the table
# already exists and you create the Table instance in this way the columns will
# be appended to the end of the existing columns.
library(synapser)
synLogin()

columns = list(
  Column(name = "my_string_column", column_type = "STRING"),
  Column(name = "my_integer_column", column_type = "INTEGER"),
  Column(name = "my_double_column", column_type = "DOUBLE"),
  Column(name = "my_boolean_column", column_type = "BOOLEAN")
)

table = Table(
  name = "my_table",
  parent_id = "syn1234",
  columns = columns
)

table |> synStore()

## Example 4: Creating a table with a named list of columns
# When specifying a number of columns via a named list setting the name attribute
# on the Column object is optional. When it is not specified it will be
# pulled from the key of the named list.
library(synapser)
synLogin()

columns = list(
  my_string_column = Column(column_type = "STRING"),
  my_integer_column = Column(column_type = "INTEGER"),
  my_double_column = Column(column_type = "DOUBLE"),
  my_boolean_column = Column(column_type = "BOOLEAN")
)

table = Table(
  name = "my_table",
  parent_id = "syn1234",
  columns = columns
)

table |> synStore()

## End(Not run)

```

Description

Configuration options for table-like entities when using the factory methods.

This dataclass controls how table-like entities (Table, Dataset, EntityView, MaterializedView, SubmissionView, VirtualTable, and DatasetCollection) are retrieved, particularly whether column schema information should be included.

Usage

```
TableOptions(include_columns=TRUE)
```

Arguments

`include_columns`

(bool) Whether to include column schema information when retrieving table-like entities. If TRUE, the returned entity will have its columns attribute populated with Column objects containing schema information (name, column_type, etc.). If FALSE, the columns attribute will be an empty named list. Including columns may result in additional API calls but provides complete table structure information. Default is TRUE.

Format

An S3-style class object.

Value

NULL

Methods

- `TableOptions(include_columns=TRUE)`: Constructor for [TableOptions](#)

Examples

```
## Not run:
# Configure table column inclusion:
library(synapser)

# Include column schema information
with_columns = TableOptions(include_columns = TRUE)

# Skip column information (faster retrieval)
without_columns = TableOptions(include_columns = FALSE)

## End(Not run)
```

Team	<i>Team</i>
------	-------------

Description

Represents a **Synapse Team**. The user-definable fields are documented in the arguments below.

Usage

Team(id=NULL, name=NULL, description=NULL, icon=NULL, can_public_join=FALSE, can_request_membership=T

Arguments

- id (Optional[int]) The ID of the team
- name (Optional[str]) The name of the team
- description (Optional[str]) A short description of the team
- icon (Optional[str]) A file handle ID for the icon image of the team
- can_public_join (Optional[bool]) TRUE if members can join without an invitation or approval
- can_request_membership (Optional[bool]) TRUE if users can create a membership request to join
- etag (Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client’s current representation of an entity is out-of-date.
- created_on (Optional[str]) The date this team was created
- modified_on (Optional[str]) The date this team was last modified
- created_by (Optional[str]) The ID of the user that created this team
- modified_by (Optional[str]) The ID of the user that last modified this team

Format

An S3-style class object.

Value

NULL

Methods

- Team(id=NULL, name=NULL, description=NULL, icon=NULL, can_public_join=FALSE, can_request_membersh etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL): Constructor for [Team](#)
- [synCreate](#)(instance, synapse_client=NULL): Creates a new team on Synapse.

- `synGetFromId(cls, id, synapse_client=NULL)`: Gets Team object using its integer id.
- `synGetFromName(cls, name, synapse_client=NULL)`: Gets Team object using its string name.

Note: You will be unable to retrieve a team by name immediately after its creation because the fragment service is eventually consistent. If you need to retrieve a team immediately following creation you should use the `synGetFromId()` function instead.

- `synGetUserMembershipStatus(instance, user_id, synapse_client=NULL)`: Retrieve a user's Team Membership Status bundle for this team.
- `synInviteToTeam(instance, user, message, force=TRUE, synapse_client=NULL)`: Invites a user to a team given the ID field on the Team instance.
- `synGetTeamMembers(instance, synapse_client=NULL)`: Gets the TeamMembers associated with a team given the ID field on the Team instance.
- `synGetOpenInvitations(instance, synapse_client=NULL)`: Gets all open invitations for a team given the ID field on the Team instance.
- `synDeleteInvitation(invitation_id, synapse_client=NULL)`: Deletes an open invitation to a team. Note: The client must be an administrator of the Team referenced by the invitation, or the invitee, to make this request.

Examples

```
## Not run:
## Example 1: Create a new team
library(synapser)
synLogin()

team = Team(
  name = "My Uniquely Named Team",
  description = "A team for my project collaborators"
) |> synCreate()

## Example 2: Get a team by ID or name
library(synapser)
synLogin()

team_by_id = Team() |> synGetFromId(id = 123456L)
team_by_name = Team() |> synGetFromName(name = "My Uniquely Named Team")

# Or fill in a Team that already has an id or a name. If both are set, the id is used.
team_by_id = Team(id = 123456L) |> synGet()
team_by_name = Team(name = "My Uniquely Named Team") |> synGet()

## Example 3: List the members of a team and invite a new one
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
members = team |> synGetTeamMembers()
for (member in members) {
  cat(sprintf("%s (admin: %s)\n", member$member$user_name, member$is_admin))
}
```

```
}

team |> synInviteToTeam(user = "my_username", message = "Please join my team!")

## Example 4: Delete a team
library(synapser)
synLogin()

team = Team() |> synGetFromId(id = 123456L)
team |> synDelete()

## End(Not run)
```

UsedEntity	<i>UsedEntity</i>
------------	-------------------

Description

Reference to a Synapse entity that was used or executed by an Activity.

Usage

UsedEntity(target_id=NULL, target_version_number=NULL)

Arguments

target_id (Optional[str]) The ID of the entity to which this reference refers.
target_version_number
 (Optional[int]) The version number of the entity to which this reference refers.

Format

An S3-style class object.

Value

NULL

Methods

- UsedEntity(target_id=NULL, target_version_number=NULL): Constructor for [UsedEntity](#)

Examples

```
## Not run:
## Recording an entity used to produce a file
library(synapser)
synLogin()

activity = Activity(
  name = "Analysis",
  used = list(UsedEntity(target_id = "syn456", target_version_number = 1L))
)
activity = activity |> synStore(parent = "syn123")

## End(Not run)
```

UsedURL	<i>UsedURL</i>
---------	----------------

Description

URL that was used or executed by an Activity such as a link to a GitHub commit or a link to a specific version of a software tool.

Usage

```
UsedURL(name=NULL, url=NULL)
```

Arguments

- | | |
|------|--|
| name | (Optional[str]) The name of the URL. |
| url | (Optional[str]) The external URL of the file that was used such as a link to a GitHub commit or a link to a specific version of a software tool. |

Format

An S3-style class object.

Value

NULL

Methods

- UsedURL(name=NULL, url=NULL): Constructor for [UsedURL](#)

Examples

```
## Not run:
## Recording a URL used to produce a file
library(synapser)
synLogin()

activity = Activity(
  name = "Analysis",
  used = list(UsedURL(name = "Reference genome", url = "https://www.ncbi.nlm.nih.gov/grc/human")),
  executed = list(UsedURL(name = "Analysis code", url = "https://github.com/my-org/my-analysis"))
)
activity = activity |> synStore(parent = "syn123")

## End(Not run)
```

UserPreference	<i>UserPreference</i>
----------------	-----------------------

Description

A UserPreference represents a user preference in the system.

Usage

```
UserPreference(name=NULL, value=NULL)
```

Arguments

name	(Optional[str]) The name of the user preference.
value	(Optional[bool]) The value of the user preference.

Format

An S3-style class object.

Value

NULL

Methods

- `UserPreference(name=NULL, value=NULL)`: Constructor for [UserPreference](#)

Examples

```
## Not run:
## Inspecting a user's preferences
# UserProfile objects are found in the `preferences` field of a UserProfile
library(synapser)
synLogin()

profile = UserProfile |> synGetFromUsername(username = "myUsername")
for (preference in profile$preferences) {
  print(sprintf("%s: %s", preference$name, preference$value))
}

## End(Not run)
```

UserProfile	<i>UserProfile</i>
-------------	--------------------

Description

UserProfile represents a user’s profile in the system.

Usage

```
UserProfile(id=NULL, username=NULL, etag=NULL, first_name=NULL, last_name=NULL, emails=list(), open_i
```

Arguments

id	(Optional[int]) A foreign key to the ID of the ‘principal’ object for the user.
username	(Optional[str]) A name chosen by the user that uniquely identifies them.
etag	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client’s current representation of an entity is out-of-date.
first_name	(Optional[str]) This person’s given name (forename)
last_name	(Optional[str]) This person’s family name (surname)
emails	(list[str]) The list of user email addresses registered to this user.
open_ids	(list[str]) The list of OpenIds bound to this user’s account.
r_studio_url	(Optional[str]) URL for RStudio server assigned to the user.
summary	(Optional[str]) A summary description about this person.
position	(Optional[str]) This person’s current position title.
location	(Optional[str]) This person’s location.
industry	(Optional[str]) The industry/discipline that this person is associated with.
company	(Optional[str]) This person’s current affiliation.

profile_picture_file_handle_id	(Optional[str]) The File Handle id of the user's profile picture.
url	(Optional[str]) A link to more information about this person.
team_name	(Optional[str]) This person's default team name.
send_email_notifications	(Optional[bool]) Should the user receive email notifications? Default TRUE.
mark_email_messages_as_read	(Optional[bool]) Should messages that are emailed to the user be marked as READ in Synapse? Default FALSE.
preferences	(Optional[list[UserPreference]]) A list of UserPreference objects.
created_on	(Optional[str]) The date this profile was created.
display_name	This field is deprecated and will always be NULL.
notification_settings	Contains a user's notification settings.

Format

An S3-style class object.

Value

NULL

Methods

- `UserProfile(id=NULL, username=NULL, etag=NULL, first_name=NULL, last_name=NULL, emails=list(), open_ids=list(), r_studio_url=NULL, summary=NULL, position=NULL, location=NULL, industry=NULL, company=NULL, profile_picture_file_handle_id=NULL, url=NULL, team_name=NULL, send_email_notifications=TRUE, mark_email_messages_as_read=FALSE, preferences=list(), created_on=NULL, display_name=NULL, notification_settings=NULL)`: Constructor for [UserProfile](#)
- `synGetFromId(cls, user_id, synapse_client=NULL)`: Gets [UserProfile](#) object using its integer id.
- `synGetFromUsername(cls, username, synapse_client=NULL)`: Gets [UserProfile](#) object using its string name.
- `synIsCertified(instance, synapse_client=NULL)`: Determine whether a user is certified.

Examples

```
## Not run:
## Getting a user's profile
library(synapser)
synLogin()

profile = UserProfile |> synGetFromUsername(username = "myUsername")
print(profile$id)
```

```

profile = UserProfile |> synGetFromId(user_id = 273948L)
print(profile$username)

# Check whether the user is certified
print(profile |> synIsCertified())

## End(Not run)

```

VirtualTable

VirtualTable

Description

A virtual table is a type of table that is dynamically built from a Synapse SQL query. Its content is read only and based off the `defining_sql` attribute. The SQL of the virtual table may NOT contain JOIN or UNION clauses and must reference a table that has a non-empty schema.

A `VirtualTable` object represents this `VirtualTable` API model in Synapse: <<https://rest-docs.synapse.org/rest/org/sagebio>>

Usage

```
VirtualTable(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=NULL)
```

Arguments

<code>id</code>	(Optional[str]) The unique immutable ID for this entity. Once issued, this ID is guaranteed to never change or be re-issued.
<code>name</code>	(Optional[str]) The name of this entity. Must be 256 characters or less. Names may only contain: letters, numbers, spaces, underscores, hyphens, periods, plus signs, apostrophes, and parentheses.
<code>parent_id</code>	(Optional[str]) The ID of the Entity that is the parent of this entity.
<code>activity</code>	(Optional[Activity]) The Activity model represents the main record of Provenance in Synapse.
<code>version_number</code>	(Optional[int]) The version number issued to this version on the object.
<code>description</code>	(Optional[str]) The description of this entity. Must be 1000 characters or less.
<code>etag</code>	(Optional[str]) Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client's current representation of an entity is out-of-date.
<code>created_on</code>	(Optional[str]) The date this entity was created.
<code>modified_on</code>	(Optional[str]) The date this entity was last modified. In YYYY-MM-DD-Thh:mm:ss.sssZ format.
<code>created_by</code>	(Optional[str]) The ID of the user that created this entity.
<code>modified_by</code>	(Optional[str]) The ID of the user that last modified this entity.

version_label	(Optional[str]) The version label for this entity.
version_comment	(Optional[str]) The version comment for this entity.
is_latest_version	(Optional[bool]) If this is the latest version of the object.
columns	(Optional[OrderedDict[str, Column]]) (Read Only) The columns of a virtual table are dynamic based on the select statement of the definingSQL. This list of columnIds is for read-only purposes.
is_search_enabled	(Optional[bool]) When creating or updating a table or view specifies if full text search should be enabled.
defining_sql	(Optional[str]) The synapse SQL statement that defines the data in the virtual table. This field may NOT contain JOIN or UNION clauses. If a JOIN or UNION clause is present, a ValueError will be raised when the store method is called.
annotations	(Optional[dict[str, Union[list[str], list[bool], list[float], list[int], list[date], list[datetime]]]]) Additional metadata associated with the virtual table. The key is the name of your desired annotations. The value is an object containing a list of values (use empty list to represent no values for key) and the value type associated with all values in the list. To remove all annotations set this to an empty named list list() or NULL and store the entity.

Format

An S3-style class object.

Value

NULL

Methods

- VirtualTable(id=NULL, name=NULL, parent_id=NULL, activity=NULL, version_number=NULL, description=NULL, etag=NULL, created_on=NULL, modified_on=NULL, created_by=NULL, modified_by=NULL, version_label=NULL, version_comment=NULL, is_latest_version=NULL, columns=list(), is_search_enabled=NULL, defining_sql=NULL, annotations=list()): Constructor for [VirtualTable](#)
- [synDeletePermissions](#)(instance, include_self=TRUE, include_container_content=FALSE, recursive=FALSE, target_entity_types=NULL, dry_run=FALSE, show_acl_details=TRUE, show_files_in_containers=TRUE, synapse_client=NULL): Delete the entire **local** Access Control List (ACL) for a given Entity. This is not scoped to a specific user or group, but rather removes all permissions associated with the Entity. After this operation, the Entity will inherit permissions from its benefactor, which is typically its parent entity or the Project it belongs to.

In order to remove permissions for a specific user or group, you should use the [synSetPermissions\(\)](#) function with the access_type set to an empty list.

By default, Entities such as FileEntity and Folder inherit their permission from their containing Project. For such Entities the Project is the Entity's 'benefactor'. This permission inheritance can be overridden by creating an ACL for the Entity. When this occurs the Entity becomes its own benefactor and all permission are determined by its own ACL.

If the ACL of an Entity is deleted, then its benefactor will automatically be set to its parent's benefactor.

Special notice for Projects: The ACL for a Project cannot be deleted, you must individually update or revoke the permissions for each user or group.

- `synGetAcl(instance, principal_id=NULL, check_benefactor=TRUE, synapse_client=NULL)`: Get the **ACL** that a user or group has on an Entity.
- `synGetPermissions(instance, synapse_client=NULL)`: Get the **permissions** that the caller has on an Entity.
- `synListAcl(instance, recursive=FALSE, include_container_content=FALSE, target_entity_types=NULL, log_tree=FALSE, synapse_client=NULL)`: List the Access Control Lists (ACLs) for this entity and optionally its children.

This function returns the local sharing settings for the entity and optionally its children. It provides a mapping of all ACLs for the given container/entity.

Important Note: This function returns the LOCAL sharing settings only, not the effective permissions that each Synapse User ID/Team has on the entities. More permissive permissions could be granted via a Team that the user has access to that has permissions on the entity, or through inheritance from parent entities.

- `synQuery(query, include_row_id_and_row_version=TRUE, convert_to_datetime=FALSE, download_location=NULL, quote_character="\"", escape_character="r\"(\)", line_end="\n", separator=",", header=TRUE, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. The results will always be returned as a data frame unless you specify a `download_location` in which case the results will be downloaded to that location. There are a number of arguments that you may pass to this function depending on if you are getting the results back as a DataFrame or downloading the results to a file.
- `synQueryPartMask(query, part_mask, timeout=250L, synapse_client=NULL)`: Query for data on a table stored in Synapse. This is a more advanced use case of the `synQuery()` function that allows you to determine what additional metadata about the table or query should also be returned. If you do not need this additional information then you are better off using the `synQuery()` function.

The query for this method uses this Rest API:

<<https://rest-docs.synapse.org/rest/POST/entity/id/table/query/async/start.html>>

- `synSetPermissions(instance, principal_id=NULL, access_type=NULL, modify_benefactor=FALSE, warn_if_inherits=TRUE, overwrite=TRUE, synapse_client=NULL)`: Sets permission that a user or group has on an Entity. An Entity may have its own ACL or inherit its ACL from a benefactor.

Examples

```
## Not run:
## Example 1: Create a new virtual table with a defining SQL query.
library(synapser)
synLogin()

virtual_table = VirtualTable(
  name = "My Virtual Table",
  description = "A test virtual table",
  parent_id = "syn12345",
  defining_sql = "SELECT * FROM syn67890"
) |> synStore()
cat("Created Virtual Table with ID:", virtual_table$id, "\n")

## Example 2: Update the defining SQL of an existing virtual table.
library(synapser)
synLogin()

virtual_table = VirtualTable(id = "syn12345") |> synGet()
virtual_table$defining_sql = "SELECT column1, column2 FROM syn67890"
virtual_table = virtual_table |> synStore()
cat("Updated Virtual Table defining SQL.\n")

## Example 3: Delete a virtual table.
library(synapser)
synLogin()

virtual_table = VirtualTable(id = "syn12345")
synDelete(virtual_table)
cat("Deleted Virtual Table.\n")

## Example 4: Query data from a virtual table.
library(synapser)
synLogin()

query_result = synQuery(query = "SELECT * FROM syn66080386")
query_result

## End(Not run)
```

WikiHeader

WikiHeader

Description

A WikiHeader contains basic metadata about a WikiPage.

Usage

```
WikiHeader(id=NULL, title=NULL, parent_id=NULL)
```


Arguments

id	(Optional[str]) The unique identifier for this wiki page.
title	(Optional[str]) The title of this page.
parent_id	(Optional[str]) When set, the WikiPage is a sub-page of the indicated parent WikiPage.

Format

An S3-style class object.

Value

NULL

Methods

- WikiHeader(id=NULL, title=NULL, parent_id=NULL): Constructor for [WikiHeader](#)

Examples

```
## Not run:
## Get the header tree (hierarchy) of wiki pages for an entity
library(synapser)
synLogin()

project = Project(name = "My uniquely named project about Alzheimer's Disease") |> synGet()
for (header in as.list(WikiHeader() |> synGet(owner_id = project$id))) {
  print(header)
}

## End(Not run)
```

WikiHistorySnapshot *WikiHistorySnapshot*

Description

A WikiHistorySnapshot contains basic information about an update to a WikiPage.

Usage

```
WikiHistorySnapshot(version=NULL, modified_on=NULL, modified_by=NULL)
```

Arguments

version	(Optional[str]) The version number of the wiki page.
modified_on	(Optional[str]) The timestamp when this version was created.
modified_by	(Optional[str]) The ID of the user that created this version.

Format

An S3-style class object.

Value

NULL

Methods

- WikiHistorySnapshot(version=NULL, modified_on=NULL, modified_by=NULL): Constructor for [WikiHistorySnapshot](#)

Examples

```
## Not run:
## Get the history of a wiki page
library(synapser)
synLogin()

project = Project(name = "My uniquely named project about Alzheimer's Disease") |> synGet()
wiki_page_id = "123456" # Replace with the ID of a wiki page in the project
for (history in as.list(WikiHistorySnapshot() |> synGet(owner_id = project$id, id = wiki_page_id))) {
  print(history)
}

## End(Not run)
```

WikiOrderHint	<i>WikiOrderHint</i>
---------------	----------------------

Description

A WikiOrderHint contains the order hint for the root wiki that corresponds to the given owner ID and type.

Usage

```
WikiOrderHint(owner_id=NULL, owner_object_type=NULL, id_list=list(), etag=NULL)
```

Arguments

- | | |
|-------------------|---|
| owner_id | (Optional[str]) The Synapse ID of the owner object (e.g., entity, evaluation, etc.). |
| owner_object_type | (Optional[str]) The type of the owner object. |
| id_list | (list[str]) The list of sub wiki ids that in the order that they should be placed relative to their siblings. |
| etag | (Optional[str]) The etag of this object. |

Format

An S3-style class object.

Value

NULL

Methods

- `WikiOrderHint(owner_id=NULL, owner_object_type=NULL, id_list=list(), etag=NULL):`
Constructor for [WikiOrderHint](#)

Examples

```
## Not run:
## Example 1: Get the WikiOrderHint for a project
# This example shows how to get a WikiOrderHint for existing wiki pages in a project.
library(synapser)
synLogin()

project = Project(name = "My uniquely named project about Alzheimer's Disease") |> synGet()
wiki_order_hint = WikiOrderHint(owner_id = project$id) |> synGet()
print(wiki_order_hint)

## Example 2: Set the WikiOrderHint for a project
# This example shows how to set a WikiOrderHint for existing wiki pages in a project.
# The WikiOrderHint is not set by default, so you need to set it explicitly.
# root_wiki_page and wiki_page_1 to wiki_page_4 are WikiPage objects that
# already exist in the project.
wiki_order_hint = WikiOrderHint(owner_id = project$id) |> synGet()

wiki_order_hint$id_list = list(
  root_wiki_page$id,
  wiki_page_1$id,
  wiki_page_3$id,
  wiki_page_2$id,
  wiki_page_4$id
)
wiki_order_hint = wiki_order_hint |> synStore()
print(wiki_order_hint)

## Example 3: Update the WikiOrderHint for a project
# This example shows how to update a WikiOrderHint for existing wiki pages in a project.
wiki_order_hint$id_list = list(
  root_wiki_page$id,
  wiki_page_1$id,
  wiki_page_2$id,
  wiki_page_3$id,
  wiki_page_4$id
)
wiki_order_hint = wiki_order_hint |> synStore()
print(wiki_order_hint)
```

End(Not run)

WikiPage	<i>WikiPage</i>
----------	-----------------

Description

Represents a **Wiki Page**.

Usage

WikiPage(id=NULL, etag=NULL, title=NULL, parent_id=NULL, markdown=NULL, attachments=list(), owner_id=

Arguments

- id (Optional[str]) The unique identifier for this wiki page.
- etag (Optional[str]) The etag of this object. Synapse employs an Optimistic Concurrency Control (OCC) scheme to handle concurrent updates. Since the E-Tag changes every time an entity is updated it is used to detect when a client’s current representation of an entity is out-of-date.
- title (Optional[str]) The title of this page.
- parent_id (Optional[str]) When set, the WikiPage is a sub-page of the indicated parent WikiPage.
- markdown (Optional[str]) The markdown content of the wiki page.
- attachments (list[dict[str, Any]]) A list of file attachments associated with the wiki page.
- owner_id (Optional[str]) The Synapse ID of the owning object (e.g., entity, evaluation, etc.).
- created_on (Optional[str]) The timestamp when this page was created.
- created_by (Optional[str]) The ID of the user that created this page.
- modified_on (Optional[str]) The timestamp when this page was last modified.
- modified_by (Optional[str]) The ID of the user that last modified this page.
- wiki_version (Optional[str]) The version number of this wiki page.
- markdown_file_handle_id (Optional[str]) The ID of the file handle containing the markdown content.
- attachment_file_handle_ids (list[str]) The list of attachment file handle ids of this page.

Format

An S3-style class object.

Value

NULL

Methods

- `WikiPage(id=NULL, etag=NULL, title=NULL, parent_id=NULL, markdown=NULL, attachments=list(), owner_id=NULL, created_on=NULL, created_by=NULL, modified_on=NULL, modified_by=NULL, wiki_version=NULL, markdown_file_handle_id=NULL, attachment_file_handle_ids=list())`: Constructor for [WikiPage](#)
- `synCopy(instance, destination_owner_id, destination_sub_page_id=NULL, update_links=TRUE, entity_map=NULL, synapse_client=NULL)`: Copy the wiki page tree of the owner entity to another entity and update internal links.
If id is set on this WikiPage, only the sub-tree rooted at that wiki page is copied. Otherwise the entire wiki of the owner entity is copied.
- `synGetAttachment(instance, file_name, download_file=TRUE, download_location=NULL, synapse_client=NULL)`: Download the wiki page attachment to a local file or return the URL.
- `synGetAttachmentHandles(instance, synapse_client=NULL)`: Get the file handles of all attachments on this wiki page.
- `synGetAttachmentPreview(instance, file_name, download_file=TRUE, download_location=NULL, synapse_client=NULL)`: Download the wiki page attachment preview to a local file or return the URL.
- `synGetMarkdownFile(instance, download_file=TRUE, download_location=NULL, synapse_client=NULL)`: Get the markdown URL of this wiki page.
- `synReformatAttachmentFileName(attachment_file_name)`: Reformat the attachment file name to be a valid attachment path.
- `synRestore(instance, synapse_client=NULL)`: Restore a specific version of a wiki page.
- `synUnzipGzippedFile(file_path)`: Unzip the gzipped file and return the file path to the unzipped file. If the file is a markdown file, the content will be printed.

Examples

```
## Not run:
## Example 1: Store a wiki page
# This example shows how to store a wiki page.
library(synapser)
synLogin()

project = Project(name = "My uniquely named project about Alzheimer's Disease") |> synGet()
wiki_page = WikiPage(owner_id = project$id, title = "My wiki page") |> synStore()
print(wiki_page)

## Example 2: Get a wiki page from Synapse
# This example shows how to get a wiki page from Synapse.
# It uses the project and wiki_page from Example 1.
wiki_page = WikiPage(owner_id = project$id, id = wiki_page$id) |> synGet()
print(wiki_page)

## Example 3: Delete a wiki page
# This example shows how to delete a wiki page.
# It uses the project and wiki_page from Example 1.
WikiPage(owner_id = project$id, id = wiki_page$id) |> synDelete()
```

```
        print("Wiki page deleted successfully.")  
## End(Not run)
```

Index

- * **Activity**
 - synDisassociateActivityFromEntity, [154](#)
 - synGetFromParent, [196](#)
- * **AgentSession**
 - synPrompt, [287](#)
 - synStart, [346](#)
 - synUpdate, [380](#)
- * **Agent**
 - synGetChatHistory, [189](#)
 - synGetSession, [236](#)
 - synPrompt, [285](#)
 - synRegister, [308](#)
 - synStartSession, [347](#)
- * **Column**
 - synList, [249](#)
- * **DatasetCollection**
 - synAddColumn, [90](#)
 - synAddItem, [97](#)
 - synDeleteColumn, [118](#)
 - synDeletePermissions, [125](#)
 - synGetAcl, [170](#)
 - synGetPermissions, [206](#)
 - synListAcl, [252](#)
 - synQuery, [290](#)
 - synQueryPartMask, [299](#)
 - synRemoveItem, [310](#)
 - synReorderColumn, [312](#)
 - synSetPermissions, [325](#)
 - synSnapshot, [340](#)
 - synUpdateRows, [386](#)
- * **Dataset**
 - synAddColumn, [88](#)
 - synAddItem, [96](#)
 - synDeleteColumn, [117](#)
 - synDeletePermissions, [123](#)
 - synGetAcl, [168](#)
 - synGetPermissions, [205](#)
 - synListAcl, [250](#)
 - synQuery, [288](#)
 - synQueryPartMask, [298](#)
 - synRemoveItem, [309](#)
 - synReorderColumn, [311](#)
 - synSetPermissions, [323](#)
 - synSnapshot, [339](#)
 - synUpdateRows, [384](#)
- * **EntityView**
 - synAddColumn, [91](#)
 - synBindSchema, [99](#)
 - synDeleteColumn, [119](#)
 - synDeletePermissions, [128](#)
 - synGetAcl, [171](#)
 - synGetPermissions, [206](#)
 - synGetSchema, [215](#)
 - synGetSchemaDerivedKeys, [223](#)
 - synListAcl, [255](#)
 - synQuery, [291](#)
 - synQueryPartMask, [301](#)
 - synReorderColumn, [313](#)
 - synSetPermissions, [326](#)
 - synSnapshot, [342](#)
 - synUnbindSchema, [371](#)
 - synUpdateRows, [388](#)
 - synValidateSchema, [395](#)
- * **Evaluation**
 - synGetAcl, [172](#)
 - synGetAllEvaluations, [182](#)
 - synGetAvailableEvaluations, [187](#)
 - synGetEvaluationsByProject, [189](#)
 - synGetPermissions, [207](#)
 - synUpdateAcl, [381](#)
- * **File**
 - synBindSchema, [101](#)
 - synChangeMetadata, [108](#)
 - synCopy, [109](#)
 - synDeletePermissions, [131](#)
 - synGetAcl, [173](#)
 - synGetFromId, [193](#)

- [synGetFromPath](#), 197
- [synGetPermissions](#), 208
- [synGetSchema](#), 217
- [synGetSchemaDerivedKeys](#), 225
- [synListAcl](#), 257
- [synSetPermissions](#), 327
- [synUnbindSchema](#), 373
- [synValidateSchema](#), 397

*** Folder**

- [synBindSchema](#), 102
- [synCopy](#), 110
- [synDeletePermissions](#), 134
- [synDeleteProjectSetting](#), 150
- [synFlattenFileList](#), 160
- [synGenerateSyncManifest](#), 162
- [synGetAcl](#), 174
- [synGetInvalidValidation](#), 199
- [synGetPermissions](#), 209
- [synGetProjectSetting](#), 213
- [synGetSchema](#), 219
- [synGetSchemaDerivedKeys](#), 227
- [synGetSchemaValidationStatistics](#), 232
- [synGetStsStorageToken](#), 237
- [synIndexFilesForMigration](#), 244
- [synListAcl](#), 259
- [synMapDirectoryToAllContainedFiles](#), 276
- [synMigrateIndexedFiles](#), 280
- [synSetPermissions](#), 328
- [synSetStorageLocation](#), 337
- [synSyncFromSynapse](#), 360
- [synSyncToSynapse](#), 369
- [synUnbindSchema](#), 374
- [synValidateSchema](#), 399
- [synWalk](#), 405

*** MaterializedView**

- [synDeletePermissions](#), 136
- [synGetAcl](#), 176
- [synGetPermissions](#), 210
- [synListAcl](#), 262
- [synQuery](#), 292
- [synQueryPartMask](#), 302
- [synSetPermissions](#), 330

*** Organization**

- [synGetAcl](#), 177
- [synGetJsonSchemas](#), 202
- [synUpdateAcl](#), 383

*** Project**

- [synBindSchema](#), 104
- [synCopy](#), 111
- [synDeletePermissions](#), 139
- [synDeleteProjectSetting](#), 151
- [synFlattenFileList](#), 161
- [synGenerateSyncManifest](#), 163
- [synGetAcl](#), 178
- [synGetInvalidValidation](#), 201
- [synGetPermissions](#), 210
- [synGetProjectSetting](#), 214
- [synGetSchema](#), 220
- [synGetSchemaDerivedKeys](#), 229
- [synGetSchemaValidationStatistics](#), 234
- [synGetStsStorageToken](#), 238
- [synIndexFilesForMigration](#), 245
- [synListAcl](#), 265
- [synMapDirectoryToAllContainedFiles](#), 277
- [synMigrateIndexedFiles](#), 282
- [synSetPermissions](#), 331
- [synSetStorageLocation](#), 338
- [synSyncFromSynapse](#), 365
- [synSyncToSynapse](#), 370
- [synUnbindSchema](#), 376
- [synValidateSchema](#), 401
- [synWalk](#), 407

*** SubmissionBundle**

- [synGetEvaluationSubmissionBundles](#), 191
- [synGetUserSubmissionBundles](#), 242

*** SubmissionStatus**

- [synBatchUpdateSubmissionStatuses](#), 98
- [synGetAllSubmissionStatuses](#), 184

*** SubmissionView**

- [synAddColumn](#), 93
- [synDeleteColumn](#), 120
- [synDeletePermissions](#), 142
- [synGetAcl](#), 179
- [synGetPermissions](#), 211
- [synListAcl](#), 267
- [synQuery](#), 294
- [synQueryPartMask](#), 303
- [synReorderColumn](#), 314
- [synSetPermissions](#), 332
- [synSnapshot](#), 343

* **Submission**

synCancel, 107
 synGetEvaluationSubmissions, 192
 synGetSubmissionCount, 240
 synGetUserSubmissions, 243

* **Table**

synAddColumn, 94
 synBindSchema, 106
 synDeleteColumn, 121
 synDeletePermissions, 145
 synDeleteRows, 152
 synGetAcl, 180
 synGetPermissions, 212
 synGetSchema, 222
 synGetSchemaDerivedKeys, 231
 synListAcl, 270
 synQuery, 295
 synQueryPartMask, 304
 synReorderColumn, 315
 synSetPermissions, 333
 synSnapshot, 345
 synStoreRows, 352
 synUnbindSchema, 378
 synUpsertRows, 390
 synValidateSchema, 403

* **Team**

synCreate, 114
 synDeleteInvitation, 122
 synGetFromId, 194
 synGetFromName, 195
 synGetOpenInvitations, 204
 synGetTeamMembers, 240
 synGetUserMembershipStatus, 241
 synInviteToTeam, 246

* **UserProfile**

synGetFromId, 194
 synGetFromUsername, 198
 synIsCertified, 247

* **VirtualTable**

synDeletePermissions, 147
 synGetAcl, 181
 synGetPermissions, 213
 synListAcl, 272
 synQuery, 297
 synQueryPartMask, 306
 synSetPermissions, 335

* **WikiPage**

synCopy, 112

synGetAttachment, 185
 synGetAttachmentHandles, 186
 synGetAttachmentPreview, 186
 synGetMarkdownFile, 203
 synReformatAttachmentFileName, 307
 synRestore, 318
 synUnzipGzippedFile, 379

Activity, 7, 8

Activity_synDisassociateActivityFromEntity
 (synDisassociateActivityFromEntity),
 154

Activity_synGetFromParent
 (synGetFromParent), 196

ActivityOptions, 9, 10, 165

Agent, 10, 11, 15

Agent_synGetChatHistory
 (synGetChatHistory), 189

Agent_synGetSession (synGetSession), 236

Agent_synPrompt (synPrompt), 285

Agent_synRegister (synRegister), 308

Agent_synStartSession
 (synStartSession), 347

AgentPrompt, 13, 13

AgentSession, 14, 15

AgentSession_synPrompt (synPrompt), 287

AgentSession_synStart (synStart), 346

AgentSession_synUpdate (synUpdate), 380

Column, 16, 17, 249

Column_synList (synList), 249

Dataset, 18, 20

Dataset_synAddColumn (synAddColumn), 88

Dataset_synAddItem (synAddItem), 96

Dataset_synDeleteColumn
 (synDeleteColumn), 117

Dataset_synDeletePermissions
 (synDeletePermissions), 123

Dataset_synGetAcl (synGetAcl), 168

Dataset_synGetPermissions
 (synGetPermissions), 205

Dataset_synListAcl (synListAcl), 250

Dataset_synQuery (synQuery), 288

Dataset_synQueryPartMask
 (synQueryPartMask), 298

Dataset_synRemoveItem (synRemoveItem),
 309

- Dataset_synReorderColumn
(synReorderColumn), [311](#)
- Dataset_synSetPermissions
(synSetPermissions), [323](#)
- Dataset_synSnapshot (synSnapshot), [339](#)
- Dataset_synUpdateRows (synUpdateRows),
[384](#)
- DatasetCollection, [25](#), [27](#)
- DatasetCollection_synAddColumn
(synAddColumn), [90](#)
- DatasetCollection_synAddItem
(synAddItem), [97](#)
- DatasetCollection_synDeleteColumn
(synDeleteColumn), [118](#)
- DatasetCollection_synDeletePermissions
(synDeletePermissions), [125](#)
- DatasetCollection_synGetAcl
(synGetAcl), [170](#)
- DatasetCollection_synGetPermissions
(synGetPermissions), [206](#)
- DatasetCollection_synListAcl
(synListAcl), [252](#)
- DatasetCollection_synQuery (synQuery),
[290](#)
- DatasetCollection_synQueryPartMask
(synQueryPartMask), [299](#)
- DatasetCollection_synRemoveItem
(synRemoveItem), [310](#)
- DatasetCollection_synReorderColumn
(synReorderColumn), [312](#)
- DatasetCollection_synSetPermissions
(synSetPermissions), [325](#)
- DatasetCollection_synSnapshot
(synSnapshot), [340](#)
- DatasetCollection_synUpdateRows
(synUpdateRows), [386](#)
- DownloadListItem, [31](#), [32](#), [155](#), [159](#)
- EntityRef, [21](#), [32](#), [33](#), [96](#), [97](#), [309](#), [310](#)
- EntityView, [33](#), [37](#)
- EntityView_synAddColumn (synAddColumn),
[91](#)
- EntityView_synBindSchema
(synBindSchema), [99](#)
- EntityView_synDeleteColumn
(synDeleteColumn), [119](#)
- EntityView_synDeletePermissions
(synDeletePermissions), [128](#)
- EntityView_synGetAcl (synGetAcl), [171](#)
- EntityView_synGetPermissions
(synGetPermissions), [206](#)
- EntityView_synGetSchema (synGetSchema),
[215](#)
- EntityView_synGetSchemaDerivedKeys
(synGetSchemaDerivedKeys), [223](#)
- EntityView_synListAcl (synListAcl), [255](#)
- EntityView_synQuery (synQuery), [291](#)
- EntityView_synQueryPartMask
(synQueryPartMask), [301](#)
- EntityView_synReorderColumn
(synReorderColumn), [313](#)
- EntityView_synSetPermissions
(synSetPermissions), [326](#)
- EntityView_synSnapshot (synSnapshot),
[342](#)
- EntityView_synUnbindSchema
(synUnbindSchema), [371](#)
- EntityView_synUpdateRows
(synUpdateRows), [388](#)
- EntityView_synValidateSchema
(synValidateSchema), [395](#)
- Evaluation, [40](#), [41](#)
- Evaluation_synGetAcl (synGetAcl), [172](#)
- Evaluation_synGetAllEvaluations
(synGetAllEvaluations), [182](#)
- Evaluation_synGetAvailableEvaluations
(synGetAvailableEvaluations),
[187](#)
- Evaluation_synGetEvaluationsByProject
(synGetEvaluationsByProject),
[189](#)
- Evaluation_synGetPermissions
(synGetPermissions), [207](#)
- Evaluation_synUpdateAcl (synUpdateAcl),
[381](#)
- File, [43](#), [46](#), [336](#)
- File_synBindSchema (synBindSchema), [101](#)
- File_synChangeMetadata
(synChangeMetadata), [108](#)
- File_synCopy (synCopy), [109](#)
- File_synDeletePermissions
(synDeletePermissions), [131](#)
- File_synGetAcl (synGetAcl), [173](#)
- File_synGetFromId (synGetFromId), [193](#)
- File_synGetFromPath (synGetFromPath),
[197](#)

- File_synGetPermissions
(synGetPermissions), 208
- File_synGetSchema (synGetSchema), 217
- File_synGetSchemaDerivedKeys
(synGetSchemaDerivedKeys), 225
- File_synListAcl (synListAcl), 257
- File_synSetPermissions
(synSetPermissions), 327
- File_synUnbindSchema (synUnbindSchema), 373
- File_synValidateSchema
(synValidateSchema), 397
- FileOptions, 47, 48, 165
- Folder, 49, 50, 336
- Folder_synBindSchema (synBindSchema), 102
- Folder_synCopy (synCopy), 110
- Folder_synDeletePermissions
(synDeletePermissions), 134
- Folder_synDeleteProjectSetting
(synDeleteProjectSetting), 150
- Folder_synFlattenFileList
(synFlattenFileList), 160
- Folder_synGenerateSyncManifest
(synGenerateSyncManifest), 162
- Folder_synGetAcl (synGetAcl), 174
- Folder_synGetInvalidValidation
(synGetInvalidValidation), 199
- Folder_synGetPermissions
(synGetPermissions), 209
- Folder_synGetProjectSetting
(synGetProjectSetting), 213
- Folder_synGetSchema (synGetSchema), 219
- Folder_synGetSchemaDerivedKeys
(synGetSchemaDerivedKeys), 227
- Folder_synGetSchemaValidationStatistics
(synGetSchemaValidationStatistics), 232
- Folder_synGetStsStorageToken
(synGetStsStorageToken), 237
- Folder_synIndexFilesForMigration
(synIndexFilesForMigration), 244
- Folder_synListAcl (synListAcl), 259
- Folder_synMapDirectoryToAllContainedFiles
(synMapDirectoryToAllContainedFiles), 276
- Folder_synMigrateIndexedFiles
(synMigrateIndexedFiles), 280
- Folder_synSetPermissions
(synSetPermissions), 328
- Folder_synSetStorageLocation
(synSetStorageLocation), 337
- Folder_synSyncFromSynapse
(synSyncFromSynapse), 360
- Folder_synSyncToSynapse
(synSyncToSynapse), 369
- Folder_synUnbindSchema
(synUnbindSchema), 374
- Folder_synValidateSchema
(synValidateSchema), 399
- Folder_synWalk (synWalk), 405
- Link, 54, 55
- LinkOptions, 55, 56, 165
- MaterializedView, 56, 58
- MaterializedView_synDeletePermissions
(synDeletePermissions), 136
- MaterializedView_synGetAcl (synGetAcl), 176
- MaterializedView_synGetPermissions
(synGetPermissions), 210
- MaterializedView_synListAcl
(synListAcl), 262
- MaterializedView_synQuery (synQuery), 292
- MaterializedView_synQueryPartMask
(synQueryPartMask), 302
- MaterializedView_synSetPermissions
(synSetPermissions), 330
- Organization, 62, 62
- Organization_synGetAcl (synGetAcl), 177
- Organization_synGetJsonSchemas
(synGetJsonSchemas), 202
- Organization_synUpdateAcl
(synUpdateAcl), 383
- Project, 63, 64, 336
- Project_synBindSchema (synBindSchema), 104
- Project_synCopy (synCopy), 111
- Project_synDeletePermissions
(synDeletePermissions), 139
- Project_synDeleteProjectSetting
(synDeleteProjectSetting), 151

- Project_synFlattenFileList
(synFlattenFileList), 161
- Project_synGenerateSyncManifest
(synGenerateSyncManifest), 163
- Project_synGetAcl (synGetAcl), 178
- Project_synGetInvalidValidation
(synGetInvalidValidation), 201
- Project_synGetPermissions
(synGetPermissions), 210
- Project_synGetProjectSetting
(synGetProjectSetting), 214
- Project_synGetSchema (synGetSchema), 220
- Project_synGetSchemaDerivedKeys
(synGetSchemaDerivedKeys), 229
- Project_synGetSchemaValidationStatistics
(synGetSchemaValidationStatistics),
234
- Project_synGetStsStorageToken
(synGetStsStorageToken), 238
- Project_synIndexFilesForMigration
(synIndexFilesForMigration),
245
- Project_synListAcl (synListAcl), 265
- Project_synMapDirectoryToAllContainedFiles
(synMapDirectoryToAllContainedFiles),
277
- Project_synMigrateIndexedFiles
(synMigrateIndexedFiles), 282
- Project_synSetPermissions
(synSetPermissions), 331
- Project_synSetStorageLocation
(synSetStorageLocation), 338
- Project_synSyncFromSynapse
(synSyncFromSynapse), 365
- Project_synSyncToSynapse
(synSyncToSynapse), 370
- Project_synUnbindSchema
(synUnbindSchema), 376
- Project_synValidateSchema
(synValidateSchema), 401
- Project_synWalk (synWalk), 407
- StorageLocation, 68, 70
- StorageLocationType, 69, 71, 72
- StoreContainerOptions, 72, 72, 349
- StoreFileOptions, 73, 73, 349
- StoreGridOptions, 74, 74, 349
- StoreJSONSchemaOptions, 75, 75, 349
- StoreTableOptions, 76, 76, 349
- Submission, 77, 78, 192, 243
- Submission_synCancel (synCancel), 107
- Submission_synGetEvaluationSubmissions
(synGetEvaluationSubmissions),
192
- Submission_synGetSubmissionCount
(synGetSubmissionCount), 240
- Submission_synGetUserSubmissions
(synGetUserSubmissions), 243
- SubmissionBundle, 79, 80, 191, 242
- SubmissionBundle_synGetEvaluationSubmissionBundles
(synGetEvaluationSubmissionBundles),
191
- SubmissionBundle_synGetUserSubmissionBundles
(synGetUserSubmissionBundles),
242
- SubmissionStatus, 80, 82
- SubmissionStatus_synBatchUpdateSubmissionStatuses
(synBatchUpdateSubmissionStatuses),
98
- SubmissionStatus_synGetAllSubmissionStatuses
(synGetAllSubmissionStatuses),
184
- SubmissionView, 83, 86
- SubmissionView_synAddColumn
(synAddColumn), 93
- SubmissionView_synDeleteColumn
(synDeleteColumn), 120
- SubmissionView_synDeletePermissions
(synDeletePermissions), 142
- SubmissionView_synGetAcl (synGetAcl),
179
- SubmissionView_synGetPermissions
(synGetPermissions), 211
- SubmissionView_synListAcl (synListAcl),
267
- SubmissionView_synQuery (synQuery), 294
- SubmissionView_synQueryPartMask
(synQueryPartMask), 303
- SubmissionView_synReorderColumn
(synReorderColumn), 314
- SubmissionView_synSetPermissions
(synSetPermissions), 332
- SubmissionView_synSnapshot
(synSnapshot), 343
- synAddColumn, 20, 21, 27, 28, 36, 37, 85, 86,
88, 90, 91, 93, 94, 352, 410, 412, 414
- synAddItem, 21, 28, 96, 97

- [synBatchUpdateSubmissionStatuses](#), [82](#), [98](#)
- [synBindSchema](#), [37](#), [46](#), [50](#), [64](#), [99](#), [101](#), [102](#), [104](#), [106](#), [412](#)
- [synCancel](#), [78](#), [107](#)
- [synChangeMetadata](#), [44](#), [46](#), [108](#)
- [synCopy](#), [46](#), [50](#), [64](#), [109–112](#), [437](#)
- [synCreate](#), [114](#), [422](#)
- [synDelete](#), [43](#), [54](#), [115](#), [410](#)
- [synDeleteColumn](#), [20](#), [21](#), [27](#), [28](#), [36](#), [37](#), [85](#), [86](#), [117–121](#), [410](#), [412](#)
- [synDeleteInvitation](#), [122](#), [423](#)
- [synDeletePermissions](#), [21](#), [28](#), [37](#), [46](#), [50](#), [58](#), [65](#), [87](#), [123](#), [125](#), [128](#), [131](#), [134](#), [136](#), [139](#), [142](#), [145](#), [147](#), [412](#), [430](#)
- [synDeleteProjectSetting](#), [51](#), [65](#), [150](#), [151](#)
- [synDeleteRows](#), [152](#), [413](#)
- [synDisassociateActivityFromEntity](#), [8](#), [34](#), [43](#), [54](#), [154](#), [410](#)
- [synDownloadListAdd](#), [155](#)
- [synDownloadListClear](#), [156](#)
- [synDownloadListFiles](#), [53](#), [67](#), [156](#), [360](#), [365](#), [369](#), [370](#)
- [synDownloadListManifest](#), [157](#)
- [synDownloadListRemove](#), [158](#)
- [synFindEntityId](#), [159](#)
- [synFlattenFileList](#), [51](#), [65](#), [160](#), [161](#)
- [synGenerateSyncManifest](#), [51](#), [65](#), [162](#), [163](#)
- [synGet](#), [46](#), [64](#), [164](#), [193](#), [197](#), [336](#), [356](#)
- [synGetAcl](#), [21](#), [28](#), [37](#), [41](#), [46](#), [51](#), [58](#), [62](#), [65](#), [87](#), [168](#), [170–174](#), [176–181](#), [381](#), [413](#), [431](#)
- [synGetAllEvaluations](#), [41](#), [182](#)
- [synGetAllSubmissionStatuses](#), [82](#), [184](#)
- [synGetAttachment](#), [185](#), [437](#)
- [synGetAttachmentHandles](#), [186](#), [437](#)
- [synGetAttachmentPreview](#), [186](#), [437](#)
- [synGetAvailableEvaluations](#), [41](#), [187](#)
- [synGetChatHistory](#), [11](#), [189](#)
- [synGetEvaluationsByProject](#), [41](#), [189](#)
- [synGetEvaluationSubmissionBundles](#), [80](#), [191](#)
- [synGetEvaluationSubmissions](#), [78](#), [192](#)
- [synGetFromId](#), [46](#), [193](#), [194](#), [195](#), [423](#), [428](#)
- [synGetFromName](#), [195](#), [423](#)
- [synGetFromParent](#), [8](#), [196](#)
- [synGetFromPath](#), [46](#), [197](#)
- [synGetFromUsername](#), [198](#), [428](#)
- [synGetInvalidValidation](#), [51](#), [65](#), [199](#), [200](#)
- [synGetJsonSchemas](#), [62](#), [202](#)
- [synGetMarkdownFile](#), [203](#), [437](#)
- [synGetOpenInvitations](#), [122](#), [204](#), [423](#)
- [synGetPermissions](#), [21](#), [28](#), [38](#), [41](#), [46](#), [51](#), [58](#), [65](#), [87](#), [205–213](#), [413](#), [431](#)
- [synGetProjectSetting](#), [51](#), [52](#), [65](#), [67](#), [213](#), [214](#), [337](#), [338](#)
- [synGetSchema](#), [38](#), [46](#), [51](#), [65](#), [215](#), [217](#), [219](#), [220](#), [222](#), [413](#)
- [synGetSchemaDerivedKeys](#), [38](#), [46](#), [51](#), [65](#), [223](#), [225](#), [227](#), [229](#), [231](#), [413](#)
- [synGetSchemaValidationStatistics](#), [51](#), [66](#), [232](#), [234](#)
- [synGetSession](#), [11](#), [236](#)
- [synGetStsStorageToken](#), [51](#), [66](#), [237](#), [238](#)
- [synGetSubmissionCount](#), [78](#), [240](#)
- [synGetTeamMembers](#), [240](#), [423](#)
- [synGetUserMembershipStatus](#), [241](#), [423](#)
- [synGetUserSubmissionBundles](#), [80](#), [242](#)
- [synGetUserSubmissions](#), [78](#), [243](#)
- [synIndexFilesForMigration](#), [51](#), [52](#), [66](#), [244](#), [245](#), [280–282](#)
- [synInviteToTeam](#), [246](#), [423](#)
- [synIsCertified](#), [247](#), [428](#)
- [synIsSynapseId](#), [248](#)
- [synList](#), [17](#), [249](#)
- [synListAcl](#), [21](#), [28](#), [38](#), [46](#), [52](#), [58](#), [66](#), [87](#), [250](#), [252](#), [254](#), [257](#), [259](#), [262](#), [265](#), [267](#), [270](#), [272](#), [413](#), [431](#)
- [synLogin](#), [274](#)
- [synLogout](#), [276](#)
- [synMapDirectoryToAllContainedFiles](#), [52](#), [66](#), [276](#), [277](#)
- [synMd5Query](#), [279](#)
- [synMigrateIndexedFiles](#), [52](#), [66](#), [244](#), [245](#), [280](#), [282](#)
- [synOnweb](#), [283](#)
- [synPrintEntity](#), [284](#)
- [synPrompt](#), [11](#), [15](#), [285](#), [287](#)
- [synQuery](#), [22](#), [29](#), [38](#), [59](#), [87](#), [88](#), [152](#), [288](#), [290–292](#), [294](#), [295](#), [297](#), [298](#), [299](#), [301–306](#), [353](#), [413](#), [415](#), [431](#)
- [synQueryPartMask](#), [22](#), [29](#), [38](#), [59](#), [88](#), [298](#), [299](#), [301–304](#), [306](#), [413](#), [431](#)
- [synReformatAttachmentFileName](#), [307](#), [437](#)
- [synRegister](#), [11](#), [308](#)
- [synRemoveItem](#), [22](#), [29](#), [309](#), [310](#)

- synReorderColumn, [20](#), [22](#), [27](#), [29](#), [36](#), [38](#), [85](#),
[86](#), [311–315](#), [410](#), [413](#)
- synRestDelete, [316](#)
- synRestGet, [317](#)
- synRestore, [318](#), [437](#)
- synRestPost, [319](#)
- synRestPut, [320](#)
- synSendMessage, [321](#)
- synSetEndpoints, [316](#), [317](#), [319](#), [320](#), [322](#)
- synSetPermissions, [21](#), [22](#), [28](#), [29](#), [37](#), [38](#),
[46](#), [47](#), [51](#), [52](#), [58](#), [65](#), [66](#), [87](#), [123](#),
[126](#), [128](#), [131](#), [134](#), [137](#), [139](#), [142](#),
[145](#), [148](#), [323](#), [325–328](#), [330–333](#),
[335](#), [412](#), [414](#), [430](#), [431](#)
- synSetProperties, [336](#)
- synSetStorageLocation, [52](#), [66](#), [337](#), [338](#)
- synSnapshot, [22](#), [29](#), [38](#), [88](#), [339](#), [340](#), [342](#),
[343](#), [345](#), [410](#), [411](#), [414](#)
- synStart, [15](#), [346](#)
- synStartSession, [11](#), [347](#)
- synStore, [21](#), [22](#), [28](#), [29](#), [37](#), [38](#), [45](#), [69](#), [86](#),
[88](#), [90](#), [91](#), [93](#), [94](#), [117–121](#),
[311–315](#), [336](#), [340–342](#), [344](#), [345](#),
[348](#), [356](#), [412](#), [413](#)
- synStoreRows, [352](#), [352](#), [391](#), [414](#), [417](#)
- synSyncFromSynapse, [52](#), [53](#), [67](#), [123](#), [134](#),
[137](#), [140](#), [143](#), [250](#), [260](#), [262](#), [265](#),
[268](#), [270](#), [273](#), [360](#), [365](#), [369](#), [370](#)
- synSyncToSynapse, [51](#), [53](#), [65](#), [67](#), [162](#), [163](#),
[369](#), [370](#)
- synUnbindSchema, [39](#), [47](#), [53](#), [67](#), [371](#), [372](#),
[374](#), [376](#), [378](#), [417](#)
- synUnzipGzippedFile, [379](#), [437](#)
- synUpdate, [15](#), [380](#)
- synUpdateAcl, [41](#), [62](#), [381](#), [383](#)
- synUpdateRows, [22](#), [29](#), [39](#), [384](#), [386](#), [388](#)
- synUpsertRows, [39](#), [352](#), [356](#), [388](#), [390](#), [414](#),
[417](#)
- synValidateSchema, [39](#), [47](#), [53](#), [67](#), [395](#), [397](#),
[399](#), [401](#), [403](#), [418](#)
- synWalk, [53](#), [67](#), [405](#), [407](#)
- Table, [410](#), [412](#)
- Table_synAddColumn (synAddColumn), [94](#)
- Table_synBindSchema (synBindSchema), [106](#)
- Table_synDeleteColumn
(synDeleteColumn), [121](#)
- Table_synDeletePermissions
(synDeletePermissions), [145](#)
- Table_synDeleteRows (synDeleteRows), [152](#)
- Table_synGetAcl (synGetAcl), [180](#)
- Table_synGetPermissions
(synGetPermissions), [212](#)
- Table_synGetSchema (synGetSchema), [222](#)
- Table_synGetSchemaDerivedKeys
(synGetSchemaDerivedKeys), [231](#)
- Table_synListAcl (synListAcl), [270](#)
- Table_synQuery (synQuery), [295](#)
- Table_synQueryPartMask
(synQueryPartMask), [304](#)
- Table_synReorderColumn
(synReorderColumn), [315](#)
- Table_synSetPermissions
(synSetPermissions), [333](#)
- Table_synSnapshot (synSnapshot), [345](#)
- Table_synStoreRows (synStoreRows), [352](#)
- Table_synUnbindSchema
(synUnbindSchema), [378](#)
- Table_synUpsertRows (synUpsertRows), [390](#)
- Table_synValidateSchema
(synValidateSchema), [403](#)
- TableOptions, [165](#), [420](#), [421](#)
- Team, [422](#), [422](#)
- Team_synCreate (synCreate), [114](#)
- Team_synDeleteInvitation
(synDeleteInvitation), [122](#)
- Team_synGetFromId (synGetFromId), [194](#)
- Team_synGetFromName (synGetFromName),
[195](#)
- Team_synGetOpenInvitations
(synGetOpenInvitations), [204](#)
- Team_synGetTeamMembers
(synGetTeamMembers), [240](#)
- Team_synGetUserMembershipStatus
(synGetUserMembershipStatus),
[241](#)
- Team_synInviteToTeam (synInviteToTeam),
[246](#)
- UsedEntity, [8](#), [424](#), [424](#)
- UsedURL, [8](#), [425](#), [425](#)
- UserPreference, [426](#), [426](#), [428](#)
- UserProfile, [427](#), [428](#)
- UserProfile_synGetFromId
(synGetFromId), [194](#)
- UserProfile_synGetFromUsername
(synGetFromUsername), [198](#)

UserProfile_synIsCertified
(synIsCertified), [247](#)

VirtualTable, [429](#), [430](#)

VirtualTable_synDeletePermissions
(synDeletePermissions), [147](#)

VirtualTable_synGetAcl (synGetAcl), [181](#)

VirtualTable_synGetPermissions
(synGetPermissions), [213](#)

VirtualTable_synListAcl (synListAcl),
[272](#)

VirtualTable_synQuery (synQuery), [297](#)

VirtualTable_synQueryPartMask
(synQueryPartMask), [306](#)

VirtualTable_synSetPermissions
(synSetPermissions), [335](#)

WikiHeader, [113](#), [432](#), [433](#)

WikiHistorySnapshot, [433](#), [434](#)

WikiOrderHint, [434](#), [435](#)

WikiPage, [436](#), [437](#)

WikiPage_synCopy (synCopy), [112](#)

WikiPage_synGetAttachment
(synGetAttachment), [185](#)

WikiPage_synGetAttachmentHandles
(synGetAttachmentHandles), [186](#)

WikiPage_synGetAttachmentPreview
(synGetAttachmentPreview), [186](#)

WikiPage_synGetMarkdownFile
(synGetMarkdownFile), [203](#)

WikiPage_synReformatAttachmentFileName
(synReformatAttachmentFileName),
[307](#)

WikiPage_synRestore (synRestore), [318](#)

WikiPage_synUnzipGzippedFile
(synUnzipGzippedFile), [379](#)