Package 'AzureTableStor'

May 6, 2026

Title Interface to the Table Storage Service in 'Azure'

Version 1.0.0

Description An interface to the table storage service in 'Azure': <https://azure.microsoft.com/en-us/services/storage/tables/>. Supplies functionality for reading and writing data stored in tables, both as part of a storage account and from a 'CosmosDB' database with the table service API. Part of the 'AzureR' family of packages.

URL https://github.com/Azure/AzureTableStor
https://github.com/Azure/AzureR

BugReports https://github.com/Azure/AzureTableStor/issues

License MIT + file LICENSE

Depends R (>= 3.3)

Imports utils, AzureRMR (>= 2.0.0), AzureStor (>= 3.0.0), jsonlite, openssl, httr, uuid, vctrs (>= 0.3.0)

Suggests testthat, knitr, rmarkdown

VignetteBuilder knitr

RoxygenNote 7.1.1

NeedsCompilation no

Author Hong Ooi [aut, cre], Microsoft [cph]

Maintainer Hong Ooi <hongooi73@gmail.com>

Repository CRAN

Date/Publication 2020-11-05 07:40:13 UTC

Contents

`create_table_operation` *Batch transactions for table storage*

Description

Batch transactions for table storage

Usage

```
create_table_operation(
  endpoint,
  path,
  options = list(),
  headers = list(),
  body = NULL,
  metadata = c("none", "minimal", "full"),
  http_verb = c("GET", "PUT", "POST", "PATCH", "DELETE", "HEAD")
)

create_batch_transaction(endpoint, operations)

do_batch_transaction(transaction, ...)

## S3 method for class 'batch_transaction'
do_batch_transaction(
  transaction,
  batch_status_handler = c("warn", "stop", "message", "pass"),
  num_retries = 10,
  ...
)
```

Arguments

`endpoint`	A table storage endpoint, of class `table_endpoint`.
`path`	The path component of the operation.
`options`	A named list giving the query parameters for the operation.
`headers`	A named list giving any additional HTTP headers to send to the host. AzureCosmosR will handle authentication details, so you don't have to specify these here.
`body`	The request body for a PUT/POST/PATCH operation.
`metadata`	The level of ODATA metadata to include in the response.
`http_verb`	The HTTP verb (method) for the operation.
`operations`	A list of individual table operation objects, each of class `table_operation`.

`transaction` For `do_batch_transaction`, an object of class `batch_transaction`.

`...` Arguments passed to lower-level functions.

`batch_status_handler`
For `do_batch_transaction`, what to do if one or more of the batch operations fails. The default is to signal a warning and return a list of response objects, from which the details of the failure(s) can be determined. Set this to "pass" to ignore the failure.

`num_retries` The number of times to retry the call, if the response is a HTTP error 429 (too many requests). The Cosmos DB endpoint tends to be aggressive at rate-limiting requests, to maintain the desired level of latency. This will generally not affect calls to an endpoint provided by a storage account.

Details

Table storage supports batch transactions on entities that are in the same table and belong to the same partition group. Batch transactions are also known as *entity group transactions*.

You can use `create_table_operation` to produce an object corresponding to a single table storage operation, such as inserting, deleting or updating an entity. Multiple such objects can then be passed to `create_batch_transaction`, which bundles them into a single atomic transaction. Call `do_batch_transaction` to send the transaction to the endpoint.

Note that batch transactions are subject to some limitations imposed by the REST API:

- All entities subject to operations as part of the transaction must have the same `PartitionKey` value.
- An entity can appear only once in the transaction, and only one operation may be performed against it.
- The transaction can include at most 100 entities, and its total payload may be no more than 4 MB in size.

Value

`create_table_operation` returns an object of class `table_operation`.

Assuming the batch transaction did not fail due to rate-limiting, `do_batch_transaction` returns a list of objects of class `table_operation_response`, representing the results of each individual operation. Each object contains elements named `status`, `headers` and `body` containing the respective parts of the response. Note that the number of returned objects may be smaller than the number of operations in the batch, if the transaction failed.

See Also

import_table_entities, which uses (multiple) batch transactions under the hood

Performing entity group transactions

Examples

```
## Not run:

endp <- table_endpoint("https://mycosmosdb.table.cosmos.azure.com:443", key="mykey")
tab <- create_storage_table(endp, "mytable")

## a simple batch insert
ir <- subset(iris, Species == "setosa")

# property names must be valid C# variable names
names(ir) <- sub("\\.", "_", names(ir))

# create the PartitionKey and RowKey properties
ir$PartitionKey <- ir$Species
ir$RowKey <- sprintf("%03d", seq_len(nrow(ir)))

# generate the array of insert operations: 1 per row
ops <- lapply(seq_len(nrow(ir)), function(i)
    create_table_operation(endp, "mytable", body=ir[i, ], http_verb="POST"))

# create a batch transaction and send it to the endpoint
bat <- create_batch_transaction(endp, ops)
do_batch_transaction(bat)


## End(Not run)
```

---

insert_table_entity *Operations on table entities (rows)*

Description

Operations on table entities (rows)

Usage

```
insert_table_entity(table, entity)

update_table_entity(
  table,
  entity,
  row_key = NULL,
  partition_key = NULL,
  etag = NULL
)

delete_table_entity(table, row_key, partition_key, etag = NULL)
```

```
list_table_entities(table, filter = NULL, select = NULL, as_data_frame = TRUE)

get_table_entity(table, row_key, partition_key, select = NULL)

import_table_entities(
  table,
  data,
  row_key = NULL,
  partition_key = NULL,
  batch_status_handler = c("warn", "stop", "message", "pass"),
  ...
)
```

Arguments

table	A table object, of class storage_table.
entity	For insert_table_entity and update_table_entity, a named list giving the properties (columns) of the entity. See 'Details' below.
row_key, partition_key	For get_table_entity, update_table_entity and delete_table_entity, the row and partition key values that identify the entity to get, update or delete. For import_table_entities, the columns in the imported data to treat as the row and partition keys. The default is to use columns named 'RowKey' and 'PartitionKey' respectively.
etag	For update_table_entity and delete_table_entity, an optional Etag value. If this is supplied, the update or delete operation will proceed only if the target entity's Etag matches this value. This ensures that an entity is only updated/deleted if it has not been modified since it was last retrieved.
filter, select	For list_table_entities, optional row filter and column select expressions to subset the result with. If omitted, list_table_entities will return all entities in the table.
as_data_frame	For list_table_entities, whether to return the results as a data frame, rather than a list of table rows.
data	For import_table_entities, a data frame. See 'Details' below.
batch_status_handler	For import_table_entities, what to do if one or more of the batch operations fails. The default is to signal a warning and return a list of response objects, from which the details of the failure(s) can be determined. Set this to "pass" to ignore the failure.
...	For import_table_entities, further named arguments passed to do_batch_transaction.

Details

These functions operate on rows of a table, also known as *entities*. insert, get, update and delete_table_entity operate on an individual row. import_table_entities bulk-inserts multiple rows of data into the table, using batch transactions. list_table_entities queries the table and returns multiple rows, subsetted on the filter and select arguments.

Table storage imposes the following requirements for properties (columns) of an entity:

- There must be properties named `RowKey` and `PartitionKey`, which together form the entity's unique identifier. These properties must be of type character.
- The property `Timestamp` cannot be used (strictly speaking, it is reserved by the system).
- There can be at most 255 properties per entity, although different entities can have different properties.
- Table properties must be atomic. In particular, they cannot be nested lists.

Note that table storage does *not* require that all entities in a table must have the same properties.

For `insert_table_entity`, `update_table_entity` and `import_table_entities`, you can also specify JSON text representing the data to insert/update/import, instead of a list or data frame.

`list_table_entities(as_data_frame=TRUE)` for a large table may be slow. If this is a problem, and you know that all entities in the table have the same schema, try setting `as_data_frame=FALSE` and converting to a data frame manually.

Value

`insert_table_entity` and `update_table_entity` return the Etag of the inserted/updated entity, invisibly.

`get_table_entity` returns a named list of properties for the given entity.

`list_table_entities` returns a data frame if `as_data_frame=TRUE`, and a list of entities (rows) otherwise.

`import_table_entities` invisibly returns a named list, with one component for each value of the `PartitionKey` column. Each component contains the results of the individual operations to insert each row into the table.

See Also

storage_table, do_batch_transaction

Understanding the table service data model

Examples

```
## Not run:

endp <- table_endpoint("https://mycosmosdb.table.cosmos.azure.com:443", key="mykey")
tab <- create_storage_table(endp, "mytable")

insert_table_entity(tab, list(
    RowKey="row1",
    PartitionKey="partition1",
    firstname="Bill",
    lastname="Gates"
))

get_table_entity(tab, "row1", "partition1")
```

```
# specifying the entity as JSON text instead of a list
update_table_entity(tab,
'{
    "RowKey": "row1",
    "PartitionKey": "partition1",
    "firstname": "Bill",
    "lastname": "Gates"
}')

# we can import to the same table as above: table storage doesn't enforce a schema
import_table_entities(tab, mtcars,
    row_key=row.names(mtcars),
    partition_key=as.character(mtcars$cyl))

list_table_entities(tab)
list_table_entities(tab, filter="firstname eq 'Satya'")
list_table_entities(tab, filter="RowKey eq 'Toyota Corolla'")

delete_table_entity(tab, "row1", "partition1")


## End(Not run)
```

`storage_table` *Operations with azure tables*

Description

Operations with azure tables

Usage

```
storage_table(endpoint, ...)

## S3 method for class 'table_endpoint'
storage_table(endpoint, name, ...)

list_storage_tables(endpoint, ...)

## S3 method for class 'table_endpoint'
list_storage_tables(endpoint, ...)

create_storage_table(endpoint, ...)

## S3 method for class 'table_endpoint'
create_storage_table(endpoint, name, ...)

## S3 method for class 'storage_table'
create_storage_table(endpoint, ...)
```

```
delete_storage_table(endpoint, ...)

## S3 method for class 'table_endpoint'
delete_storage_table(endpoint, name, confirm = TRUE, ...)

## S3 method for class 'storage_table'
delete_storage_table(endpoint, ...)
```

Arguments

endpoint	An object of class table_endpoint or, for create_storage_table.storage_table, an object of class storage_table.
...	Other arguments passed to lower-level functions.
name	The name of a table in a storage account.
confirm	For deleting a table, whether to ask for confirmation.

Details

These methods are for accessing and managing tables within a storage account.

Value

storage_table and create_storage_table return an object of class storage_table. list_storage_tables returns a list of such objects.

See Also

table_endpoint, table_entity

Examples

```
## Not run:

endp <- table_endpoint("https://mystorageacct.table.core.windows.net", key="mykey")

create_storage_table(endp, "mytable")
tab <- storage_table(endp, "mytable2")
create_storage_table(tab)
list_storage_tables(endp)
delete_storage_table(tab)
delete_storage_table(endp, "mytable")


## End(Not run)
```

table_endpoint *Table storage endpoint*

Description

Table storage endpoint object, and method to call it.

Usage

```
table_endpoint(
  endpoint,
  key = NULL,
  token = NULL,
  sas = NULL,
  api_version = getOption("azure_storage_api_version")
)

call_table_endpoint(
  endpoint,
  path,
  options = list(),
  headers = list(),
  body = NULL,
  ...,
  http_verb = c("GET", "DELETE", "PUT", "POST", "HEAD", "PATCH"),
  http_status_handler = c("stop", "warn", "message", "pass"),
  return_headers = (http_verb == "HEAD"),
  metadata = c("none", "minimal", "full"),
  num_retries = 10
)
```

Arguments

endpoint	For table_endpoint, the URL of the table service endpoint. This will be of the form https://{account-name}.table.core.windows.net if the service is provided by a storage account in the Azure public cloud, while for a CosmosDB database, it will be of the form https://{account-name}.table.cosmos.azure.com:443. For call_table_endpoint, an object of class table_endpoint.
key	The access key for the storage account.
token	An Azure Active Directory (AAD) authentication token. For compatibility with AzureStor; not used for table storage.
sas	A shared access signature (SAS) for the account. At least one of key or sas should be provided.
api_version	The storage API version to use when interacting with the host. Defaults to "2019-07-07".

path	For call_table_endpoint, the path component of the endpoint call.
options	For call_table_endpoint, a named list giving the query parameters for the operation.
headers	For call_table_endpoint, a named list giving any additional HTTP headers to send to the host. AzureCosmosR will handle authentication details, so you don't have to specify these here.
body	For call_table_endpoint, the request body for a PUT/POST/PATCH call.
...	For call_table_endpoint, further arguments passed to AzureStor::call_storage_endpoint and httr::VERB.
http_verb	For call_table_endpoint, the HTTP verb (method) of the operation.
http_status_handler	For call_table_endpoint, the R handler for the HTTP status code of the response. "stop", "warn" or "message" will call the corresponding handlers in httr, while "pass" ignores the status code. The latter is primarily useful for debugging purposes.
return_headers	For call_table_endpoint, whether to return the (parsed) response headers instead of the body. Ignored if http_status_handler="pass".
metadata	For call_table_endpoint, the level of ODATA metadata to include in the response.
num_retries	The number of times to retry the call, if the response is a HTTP error 429 (too many requests). The Cosmos DB endpoint tends to be aggressive at rate-limiting requests, to maintain the desired level of latency. This will generally not affect calls to an endpoint provided by a storage account.

Value

table_endpoint returns an object of class table_endpoint, inheriting from storage_endpoint. This is the analogue of the blob_endpoint, file_endpoint and adls_endpoint classes provided by the AzureStor package.

call_table_endpoint returns the body of the response by default, or the headers if return_headers=TRUE. If http_status_handler="pass", it returns the entire response object without modification.

See Also

storage_table, table_entity, AzureStor::call_storage_endpoint

Table service REST API reference

Authorizing requests to Azure storage services

Examples

```
## Not run:

# storage account table endpoint
table_endpoint("https://mystorageacct.table.core.windows.net", key="mykey")

# Cosmos DB table endpoint
```

```
table_endpoint("https://mycosmosdb.table.cosmos.azure.com:443", key="mykey")

## End(Not run)
```

Index