jeremydaly / dynamodb-toolbox

A simple set of tools for working with Amazon DynamoDB and the DocumentClient

dynamodbtoolbox.com

MIT License

610 stars 55 forks

Star Watch

Code Issues 27 Pull requests 4 Actions Projects Wiki

master

jeremydaly minor merge fixes ... 7 days ago 208

View code

README.md

DynamoDB Toolbox

build passing npm v0.3.0 license MIT coverage 83%

Single Table Designs have never been this easy!

The **DynamoDB Toolbox** is a set of tools that makes it easy to work with Amazon DynamoDB and the DocumentClient. It's designed with **Single Tables** in mind, but works just as well with multiple tables. It lets you define your Entities (with typings and aliases) and map them to your DynamoDB tables. You can then **generate the API parameters** to `put`, `get`, `delete`, `update`, `query`, `scan`, `batchGet`, and `batchWrite` data by passing in JavaScript objects. The DynamoDB Toolbox will map aliases, validate and coerce types, and even write complex `UpdateExpression`s for you. 😉

Installation and Basic Usage

Install the DynamoDB Toolbox with npm:

```
npm i dynamodb-toolbox
```

Require or import `Table` and `Entity` from `dynamodb-toolbox`:

```
const { Table, Entity } = require('dynamodb-toolbox')
```

Create a Table (with the DocumentClient):

```
// Require AWS SDK and instantiate DocumentClient
const DynamoDB = require('aws-sdk/clients/dynamodb')
const DocumentClient = new DynamoDB.DocumentClient()

// Instantiate a table
const MyTable = new Table({
  // Specify table name (used by DynamoDB)
  name: 'my-table',

  // Define partition and sort keys
  partitionKey: 'pk',
  sortKey: 'sk',

  // Add the DocumentClient
  DocumentClient
})
```

Create an Entity:

```
const Customer = new Entity({
  // Specify entity name
  name: 'Customer',
```

```
  // Define attributes
  attributes: {
    id: { partitionKey: true }, // flag as partitionKey
    sk: { hidden: true, sortKey: true }, // flag as sortKey and mark hidden
    name: { map: 'data' }, // map 'name' to table attribute 'data'
    co: { alias: 'company' }, // alias table attribute 'co' to 'company'
    age: { type: 'number' }, // set the attribute type
    status: ['sk',0], // composite key mapping
    date_added: ['sk',1] // composite key mapping
  },

  // Assign it to our table
  table: MyTable
})
```

Put an item:

```
// Create my item (using table attribute names or aliases)
let item = {
  id: 123,
  name: 'Jane Smith',
  company: 'ACME',
  age: 35,
  status: 'active',
  date_added: '2020-04-24'
}

// Use the 'put' method of Customer
let result = await Customer.put(item)
```

The item will be saved to DynamoDB like this:

```
{
  "pk": 123,
  "sk": "active#2020-04-24",
  "data": "Jane Smith",
  "co": "ACME",
  "age": 35
}
```

You can then get the data:

```
// Specify my item
let item = {
```

```
  id: 123,
  status: 'active',
  date_added: '2019-04-24'
}

// Use the 'get' method of Customer
let response = await Customer.get(item)
```

This will return the object mapped to your aliases and composite key mappings:

```
{
  Item: {
    id: 123,
    name: 'Jane Smith',
    company: 'ACME',
    age: 35,
    status: 'active',
    date_added: '2020-04-24'
  }
}
```

This is *NOT* an ORM (at least it's not trying to be)

There are several really good Object-Relational Mapping tools (ORMs) out there for DynamoDB. There's the Amazon DynamoDB DataMapper For JavaScript, @Awspilot's DynamoDB project, @baseprime's dynamodb package, and many more.

If you like working with ORMs, that's great, and you should definitely give these projects a look. But personally, I really dislike ORMs (especially ones for relational databases). I typically find them cumbersome and likely to generate terribly inefficient queries (you know who you are). So this project is not an ORM, or at least it's not trying to be. This library helps you generate the necessary parameters needed to interact with the DynamoDB API by giving you a **consistent interface** and **handling all the heavy lifting** when working with the DynamoDB API. For convenience, this library will call the DynamoDB API for you and automatically parse the results, but you're welcome to just let it generate all (or just some) of the parameters for you. Hopefully this library will make the vast majority of your DynamoDB interactions super simple, and maybe even a little bit fun! 😎

Features

- **Table Schemas and DynamoDB Typings:** Define your Table and Entity data models using a simple JavaScript object structure, assign DynamoDB data types, and

optionally set defaults.

- **Magic UpdateExpressions:** Writing complex `UpdateExpression` strings is a major pain, especially if the input data changes the underlying clauses or requires dynamic (or nested) attributes. This library handles everything from simple `SET` clauses, to complex `list` and `set` manipulations, to defaulting values with smartly applied `if_not_exists()` to avoid overwriting data.
- **Bidirectional Mapping and Aliasing:** When building a single table design, you can define multiple entities that map to the same table. Each entity can reuse fields (like `pk` and `sk`) and map them to different aliases depending on the item type. Your data is automatically mapped correctly when reading and writing data.
- **Composite Key Generation and Field Mapping:** Doing some fancy data modeling with composite keys? Like setting your `sortKey` to `[country]#[region]#[state]#[county]#[city]#[neighborhood]` model hierarchies? DynamoDB Toolbox lets you map data to these composite keys which will both autogenerate the value *and* parse them into fields for you.
- **Type Coercion and Validation:** Automatically coerce values to strings, numbers and booleans to ensure consistent data types in your DynamoDB tables. Validate `list`, `map`, and `set` types against your data. Oh yeah, and `set`s are automatically handled for you. 😉
- **Powerful Query Builder:** Specify a `partitionKey`, and then easily configure your sortKey conditions, filters, and attribute projections to query your primary or secondary indexes. This library can even handle pagination with a simple `.next()` method.
- **Simple Table Scans:** Scan through your table or secondary indexes and add filters, projections, parallel scans and more. And don't forget the pagination support with `.next()`.
- **Filter and Condition Expression Builder:** Build complex Filter and Condition expressions using a standardized `array` and `object` notation. No more appending strings!
- **Projection Builder:** Specify which attributes and paths should be returned for each entity type, and automatically filter the results.
- **Secondary Index Support:** Map your secondary indexes (GSIs and LSIs) to your table, and dynamically link your entity attributes.
- **Batch Operations:** Full support for batch operations with a simpler interface to work with multiple entities and tables.
- **Transactions:** Full support for transaction with a simpler interface to work with multiple entities and tables.

- **Default Value Dependency Graphs:** Create dynamic attribute defaults by chaining other dynamic attribute defaults together.
- **TypeScript Support:** v0.3 of this library was rewritten in TypeScript to provide strong typing support. Additional work is still required to support schema typing.

Table of Contents

Conventions, Motivations, and Migrations from v0.1

One of the most important goals of this library is to be as **unopinionated** as possible, giving you the flexibility to bend it to your will and build amazing applications. But another important goal is developer efficiency and ease of use. In order to balance these two goals, some assumptions had to be made. These include the "default" behavior of the library (all of which, btw, can be disabled with a simple configuration change). If you are using v0.1, you'll notice **a lot** of changes.

- **`autoExecute` and `autoParse` are enabled by default.** The original version of this library only handled limited "parameter generation", so it was necessary for you to pass the payloads to the `DocumentClient`. The library now provides support for all API options for each supported method, so by default, it will make the DynamoDB API call and parse the results, saving you redundant code. If you'd rather it didn't do this, you can disable it.
- **It assumes a Single Table DynamoDB design.** Watch the Rick Houlihan videos and read Alex DeBrie's book. The jury is no longer out on this: Single Table designs are what all the cool kids are doing. This library assumes that you will have multiple "Entities" associated with a single "Table", so this requires you to instantiate a `Table` and add at least one `Entity` to it. If you have multiple `Table`s and just one `Entity` type per `Table`, that's fine, it'll still make your life much easier. Also, `batchGet` and `batchWrite` support multiple tables, so we've got you covered.
- **Entity Types are added to all items.** Since this library assumes a Single Table design, it needs a way to reliably distinguish between Entity types. It does this by adding an "Entity Type" field to each item in your table. v0.1 used `__model`, but this has been changed to `_et` (short for "Entity Type"). Don't like this? Well, you can either disable it completely (but the library won't be able to parse entities into their aliases for you), or change the attribute name to something more snappy. It is purposefully short to minimize table storage (because item storage size includes the

attribute names). Also, by default, Entities will alias this field to `entity` (but you can change that too).

- **Created and modified timestamps are enabled by default.** I can't think of many instances where created and modified timestamps aren't used in database records, so the library now automatically adds `_ct` and `_md` attributes when items are `put` or `update` d. Again, these are kept purposefully short. You can disable them, change them, or even implement them yourself if you really want. By default, Entities will alias these attributes to `created` and `modified` (customizable, of course), and will automatically apply an `if_not_exists()` on updates so that the `created` date isn't overwritten.
- **Option names have been shortened using camelCase.** Nothing against long and descriptive names, but typing `ReturnConsumedCapacity` over and over again just seems like extra work. For simplification purposes, all API request parameters have been shortened to things like `capacity`, `consistent` and `metrics`. The documentation shows which parameter they map to, but they should be intuitive enough to guess.
- **All configurations and options are plain JavaScript `objects`.** There are lots of JS libraries that use function chaining (like `table.query('some pk value').condition('some condition').limit(50)`). I really like this style for lots of use cases, but it **just feels wrong** to me when using DynamoDB. DynamoDB is the OG of cloud native databases. It's configured using IaC and its API is HTTP-based and uses structured JSON, so writing queries and other interactions using its native format just seems like the right thing to do. IMO, this makes your code more explicit and easier to reason about. Your `options` could actually be stored as JSON and (unless you're using functions to define defaults on Entity attributes) your Table and Entity configurations could be too.
- **API responses match the DynamoDB API responses.** Something else I felt strongly about was the response signature returned by the library's methods. The DynamoDB Toolbox is a tool to help you interact with the DynamoDB API, **NOT a replacement for it**. ORMs typically trade ease of use with a tremendous amount of lock-in. But at the end of the day, it's just generating queries (and probably bad ones at that). DynamoDB Toolbox provides a number of helpful features to make constructing your API calls easier and more consistent, but the exact payload is always available to you. You can rip out this library whenever you want and just use the raw payloads if you really wanted to. This brings us to the responses. Other than aliasing the `Items` and `Attributes` returned from DynamoDB, the structure and format of the responses is the **exact same** (including any other meta data returned). This not only makes the library (kind of) future proof, but also allows you to reuse or repurpose any code or tools you've already written to deal with API responses.

- **Attributes with NULL values are removed (by default).** This was a hard one. I actually ran a Twitter poll to see how people felt about this, and although the reactions were mixed, *"Remove the attributes"* came out on top. I can understand the use cases for `NULL`s, but since NoSQL database attribute names are part of the storage considerations, it seems more logical to simply check for the absence of an attribute, rather than a `NULL` value. You may disagree with me, and that's cool. I've provided a `removeNullAttributes` table setting that allows you to disable this and save `NULL` attributes to your heart's content. I wouldn't, but the choice is yours.

Hopefully these all make sense and will make working with the library easier.

Tables

Tables represent one-to-one mappings to your DynamoDB tables. They contain information about your table's name, primary keys, indexes, and more. They are also used to organize and coordinate operations between **entities**. Tables support a number of methods that allow you to interact with your entities including performing **queries**, **scans**, **batch gets** and **batch writes**.

To define a new table, import it into your script:

```
const { Table } = require('dynamodb-toolbox')
```

Then create a new `Table` instance by passing in a valid `Table` definition.

```
const MyTable = new Table({
  ... table definition...
})
```

Specifying Table Definitions

`Table` takes a single parameter of type `object` that accepts the following properties:

Property	Type	Required	Description
name	`string`	yes	The name of your DynamoDB table (this will be used as the `TableName` property)

Property	Type	Required	Description
alias	`string`	no	An optional alias to reference your table when using "batch" features
partitionKey	`string`	yes	The attribute name of your table's partitionKey
sortKey	`string`	no	The attribute name of your table's sortKey
entityField	`boolean` or `string`	no	Disables or overrides entity tracking field name (default: `_et`)
attributes	`object`	no	Complex type that optionally specifies the name and type of each attributes (see below)
indexes	`object`	no	Complex type that optionally specifies the name keys of your secondary indexes (see below)
autoExecute	`boolean`	no	Enables automatic execution of the DocumentClient method (default: `true`)
autoParse	`boolean`	no	Enables automatic parsing of returned data when `autoExecute` is `true` (default: `true`)
removeNullAttributes	`boolean`	no	Removes null attributes instead of setting them to `null` (default: `true`)
DocumentClient	`DocumentClient`	*	A valid instance of the AWS DocumentClient

** A Table can be instantiated without a DocumentClient, but most methods require it before execution*

Table Attributes

The Table `attributes` property is an `object` that specifies the *names* and *types* of attributes associated with your DynamoDB table. This is an optional input that allows you to control attribute types. If an `Entity` object contains an attribute with the same name, but a different type, an error will be thrown. Each key in the object represents the **attribute name** and the value represents its DynamoDB **type**.

```
attributes: {
  pk: 'string',
  sk: 'number',
  attr1: 'list',
  attr2: 'map',
  attr3: 'boolean',
  ...
}
```

Valid DynamoDB types are: `string`, `boolean`, `number`, `list`, `map`, `binary`, or `set`.

Table Indexes

The `indexes` property is an `object` that specifies the *names* and *keys* of the secondary indexes on your DynamoDB table. Each key represents the **index name** and its value must contain an object with a `partitionKey` AND/OR a `sortKey`. `partitionKey`s and `sortKey`s require a value of type `string` that references an table attribute. If you use the same `partitionKey` as the table's `partitionKey`, or you only specify a `sortKey`, the library will recognize them as Local Secondary Indexes (LSIs). Otherwise, they will be Global Secondary Indexes (GSIs).

```
indexes: {
  GSI1: { partitionKey: 'GSI1pk', sortKey: 'GSI1sk' },
  GSI2: { partitionKey: 'test' },
  LSI1: { partitionKey: 'pk', sortKey: 'other_sk' },
  LSI2: { sortKey: 'data' }
}
```

NOTE: The **index name** must match the index name on your table as it will be used in queries and other operations. The index must include the table's `entityField` attribute for automatic parsing of returned data.

Entities

An **Entity** represent a well-defined schema for a DynamoDB item. An Entity can represent things like a *User*, an *Order*, an *Invoice Line Item*, a *Configuration Object*, or whatever else you want. Each `Entity` defined with the DynamoDB Toolbox must be attached to a `Table`. An `Entity` defines its own attributes, but can share these attributes with other entities on the same table (either explicitly or coincidentally). Entities must flag an attribute as a `partitionKey` and if enabled on the table, a `sortKey` as well.

Note that a `Table` can have multiple Entities, but an `Entity` can only have one `Table`.

To define a new entity, import it into your script:

```
const { Entity } = require('dynamodb-toolbox')
```

Then create a new `Entity` instance by passing in a valid `Entity` definition.

```
const MyEntity = new Entity({
  ... entity definition...
})
```

Specifying Entity Definitions

`Entity` takes a single parameter of type `object` that accepts the following properties:

Property	Type	Required	Description
name	`string`	yes	The name of your entity (must be unique to its associated `Table`)
timestamps	`boolean`	no	Automatically add and manage *created* and *modified* attributes
created	`string`	no	Override default *created* attribute name (default: `_ct`)
modified	`string`	no	Override default *modified* attribute name (default: `_md`)
createdAlias	`string`	no	Override default *created* alias name (default: `created`)

Property	Type	Required	Description
modifiedAlias	`string`	no	Override default *modified* alias name (default: `modified`)
typeAlias	`string`	no	Override default *entity type* alias name (default: `entity`)
attributes	`object`	yes	Complex type that specifies the schema for the entity (see below)
autoExecute	`boolean`	no	Enables automatic execution of the DocumentClient method (default: *inherited from Table*)
autoParse	`boolean`	no	Enables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Table*)
table	`Table`	*	A valid `Table` instance

* *An Entity can be instantiated without a `table`, but most methods require one before execution*

Entity Attributes

The `attributes` property is an `object` that represents the attribute names, types, and other properties related to each attribute. Each key in the object represents the **attribute name** and the value represents its properties. The value can be a `string` that represents the DynamoDB type, an `object` that allows for additional configurations, or an `array` that maps to composite keys.

Using a `string`

Attributes can be defined using only a `string` value that corresponds to a DynamoDB type.

```
schema: {
  attr1: 'string',
  attr2: 'number',
  attr3: 'list',
  attr4: 'map',
  ...
}
```

Valid types are: `string`, `boolean`, `number`, `list`, `map`, `binary`, or `set`.

Using an `object`

For more control over an attribute's behavior, you can specify an object as the attribute's value. Some options are specific to certain types. The following properties and options are available, all of which are optional:

Property	Type	For Types	Description
type	`String`	all	The DynamoDB type for this attribute. Valid values are `string`, `boolean`, `number`, `list`, `map`, `binary`, or `set`. Defaults to `string`.
coerce	`boolean`	`string`, `boolean`, `number`, `list`	Coerce values to the specified type. Enabled by default on `string`, `boolean`, and `number`. If enabled on `list` types, the interpreter will try to split a string by commas.
default	*same as* `type` or `function`	all	Specifies a default value (if none provided) when using `put` or `update`. This also supports functions for creating custom default. See more below.
dependsOn	`string` or `array` of `string`s	all	Creates a dependency graph for default values. For example, if the attribute uses a default value that requires another attribute's default value, this will ensure dependent attributes' default values are calcuated first.
onUpdate	`boolean`	all	Forces `default` values to be passed on every `update`.
hidden	`boolean`	all	Hides attribute from returned JavaScript object when auto-parsing is enabled or when using the `parse` method.

Property	Type	For Types	Description
required	`boolean` or "always"	all	Specifies whether an attribute is required. A value of `true` requires the attribute for all `put` operations. A `string` value of "always" requires the attribute for `put` *and* `update` operations.
alias	`string`	all	Adds a bidirectional alias to the attribute. All input methods can use either the attribute name or the alias when passing in data. Auto-parsing and the `parse` method will map attributes to their alias.
map	`string`	all	The inverse of the `alias` option, allowing you to specify your alias as the key and map it to an attribute name.
setType	`string`	`set`	Specifies the type for `set` attributes. Allowed values are `string`, `number`, `binary`
delimiter	`string`	*composite keys*	Specifies the delimiter to use if this attribute stores a composite key (see Using an `array` for composite keys)
prefix	`string`	`string`	A prefix to be added to an attribute when saved to DynamoDB. This prefix will be removed when parsing the data.
suffix	`string`	`string`	A suffix to be added to an attribute when saved to DynamoDB. This suffix will be removed when parsing the data.
transform	`function`	all	A function that transforms the input before sending to DynamoDB. This accepts two arguments, the value passed and an object containing the data from other attributes.

Property	Type	For Types	Description
partitionKey	`boolean` or `string`	all	Flags an attribute as the 'partitionKey' for this Entity. If set to `true`, it will be mapped to the Table's `partitionKey`. If set to the name of an **index** defined on the Table, it will be mapped to the secondary index's `partitionKey`
sortKey	`boolean` or `string`	all	Flags an attribute as the 'sortKey' for this Entity. If set to `true`, it will be mapped to the Table's `sortKey`. If set to the name of an **index** defined on the Table, it will be mapped to the secondary index's `sortKey`

NOTE: One attribute *must* be set as the `partitionKey`. If the table defines a `sortKey`, one attribute *must* be set as the `sortKey`. Assignment of secondary indexes is optional. If an attribute is used across multiple indexes, an `array` can be used to specify multiple values.

Example:

```
attributes: {
  user_id: { partitionKey: true },
  sk: { type: 'number', hidden: true, sortKey: true },
  data: { coerce: false, required: true, alias: 'name' },
  departments: { type: 'set', setType: 'string', map: 'dept' },
  ...
}
```

Using an `array` for composite keys

NOTE: The interface for composite keys may be changing in v0.2 to make it easier to customize.

Composite keys in DynamoDB are incredibly useful for creating hierarchies, one-to-many relationships, and other powerful querying capabilities (see here). The DynamoDB Toolbox lets you easily work with composite keys in a number of ways. In some cases, there is no need to store the data in the same record twice if you are already combining it into a single attribute. By using composite key mappings, you can store data together in a single field, but still be able to structure input data *and* parse the output into separate attributes.

The basic syntax is to specify an `array` with the mapped attribute name as the first element, and the index in the composite key as the second element. For example:

```
attributes: {
  user_id: { partitionKey: true  },
  sk: { hidden: true, sortKey: true },
  status: ['sk',0],
  date: ['sk',1],
  ...
}
```

This maps the `status` and `date` attributes to the `sk` attribute. If a `status` and `date` are supplied, they will be combined into the `sk` attribute as `[status]#[date]`. When the data is retrieved, the `parse` method will automatically split the `sk` attribute and return the values with `status` and `date` keys. By default, the values of composite keys are stored as separate attributes, but that can be changed by adding in an option configuration as the third array element.

Passing in a configuration Composite key mappings are `string`s by default, but can be overridden by specifying either `string`, `number`, or `boolean` as the third element in the array. Composite keys are automatically coerced into `string`s, so only the aforementioned types are allowed. You can also pass in a configuration `object` as the third element. This uses the same configuration properties as above. In addition to these properties, you can also specify a `boolean` property of `save`. This will write the value to the mapped composite key, but also add a separate attribute that stores the value.

```
attributes: {
  user_id: { partitionKey: true  },
  sk: { hidden: true, sortKey: true },
  status: ['sk',0, { type: 'boolean', save: false, default: true }],
  date: ['sk',1, { required: true }],
  ...
}
```

Customize defaults with a `function`

In simple situations, defaults can be static values. However, for advanced use cases, you can specify an anonymous function to dynamically calculate the value. The function takes a single argument that contains an object of the inputed data (including aliases). This opens up a number of really powerful use cases:

Generate the current date and time:

```
attributes: {
  user_id: { partitionKey: true },
  created: { default: () => new Date().toISOString() },
  ...
}
```

Generate a custom composite key:

```
attributes: {
  user_id: { partitionKey: true  },
  sk: { sortKey: true, default: (data) => `sort-${data.status}|${data.date_ad
  status: 'boolean',
  date_added: 'string'
  ...
}
```

Create conditional defaults:

```
attributes: {
  user_id: { partitionKey: true  },
  sk: {
    sortKey: true,
    default: (data) => {
      if (data.status && data.date_added) {
        return data.date_added
      } else {
        return null // field will not be defaulted
      }
    }
  },
  status: 'boolean',
  date_added: 'string'
  ...
}
```

Table Properties

get/set `DocumentClient`

The `DocumentClient` property allows you to get reference to the table's assigned `DocumentClient`, or to add/update the table's `DocumentClient`. When setting this property, it must be a valid instance of the AWS DocumentClient.

get/set `entities`

The `entities` property is used to add entities to the table. When adding entities, property accepts either an `array` of `Entity` instances, or a single `Entity` instance. This will add the entities to the table and create a table property with the same name as your entities `name`. For example, if an entity with the name `User` is assigned: `MyTable.entities = User`, then the Entity and its properties and methods will be accessible via `MyTable.User`.

The `entities` property will retrieve an `array` of `string`s containing all entity names attached to the table.

get/set `autoExecute`

This property will retrieve a `boolean` indicating the current `autoExecute` setting on the table. You can change this setting by supplying a `boolean` value.

get/set `autoParse`

This property will retrieve a `boolean` indicating the current `autoParse` setting on the table. You can change this setting by supplying a `boolean` value.

Table Methods

query(partitionKey [,options] [,parameters])

> The Query operation finds items based on primary key values. You can query any table or secondary index that has a composite primary key (a partition key and a sort key).

The `query` method is a wrapper for the DynamoDB Query API. The DynamoDB Toolbox `query` method supports all **Query** API operations. The `query` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `queryParams` can be used, but will only retrieve the generated parameters.

The `query()` method accepts three arguments. The first argument is used to specify the `partitionKey` you wish to query against (KeyConditionExpression). The value must match the type of your table's partition key.

The second argument is an `options` object that specifies the details of your query. The following options are all optional (corresponding Query API references in parentheses):

Option	Type	Description
index	`string`	Name of secondary index to query. If not specified, the query executes on the primary index. The index must include the table's `entityField` attribute for automatic parsing of returned data. (IndexName)
limit	`number`	The maximum number of items to retrieve per query. (Limit)
reverse	`boolean`	Reverse the order or returned items. (ScanIndexForward)
consistent	`boolean`	Enable a consistent read of the items (ConsistentRead)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
select	`string`	The attributes to be returned in the result. One of either `string`
eq	same as `sortKey`	Specifies `sortKey` condition to be *equal* to supplied value. (KeyConditionExpression)
lt	same as `sortKey`	Specifies `sortKey` condition to be *less than* supplied value. (KeyConditionExpression)
lte	same as `sortKey`	Specifies `sortKey` condition to be *less than or equal to* supplied value. (KeyConditionExpression)
gt	same as `sortKey`	Specifies `sortKey` condition to be *greater than* supplied value. (KeyConditionExpression)
gte	same as `sortKey`	Specifies `sortKey` condition to be *greater than or equal to* supplied value. (KeyConditionExpression)
between	`array`	Specifies `sortKey` condition to be *between* the supplied values. Array should have two values matching the `sortKey` type. (KeyConditionExpression)
beginsWith	same as `sortKey`	Specifies `sortKey` condition to *begin with* the supplied values. (KeyConditionExpression)
filters	`array` or `object`	A complex `object` or `array` of objects that specifies the query's filter condition. See Filters and Conditions. (FilterExpression)

Option	Type	Description
attributes	`array` or `object`	An `array` or array of complex `objects` that specify which attributes should be returned. See Projection Expression below (ProjectionExpression)
startKey	`object`	An object that contains the `partitionKey` and `sortKey` of the first item that this operation will evaluate. (ExclusiveStartKey)
entity	`string`	The name of a table Entity to evaluate `filters` and `attributes` against.
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

```
let result = await MyTable.query(
  'user#12345', // partition key
  {
    limit: 50, // limit to 50 items
    beginsWith: 'order#', // select items where sort key begins with value
    reverse: true, // return items in descending order (newest first)
    capacity: 'indexes', // return the total capacity consumed by the indexes
    filters: { attr: 'total', gt: 100 }, // only show orders above $100
    index: 'GSI1' // query the GSI1 secondary index
  }
)
```

Return Data

The data is returned with the same response syntax as the DynamoDB Query API. If `autoExecute` and `autoParse` are enabled, any `Items` data returned will be parsed into its corresponding Entity's aliases. Otherwise, the DocumentClient will return the unmarshalled data. If the response is parsed by the library, a `.next()` method will be available on the returned object. Calling this function will call the `query` method again using the same parameters and passing the `LastEvaluatedKey` in as the `ExclusiveStartKey`. This is a convenience method for paginating the results.

scan([options] [,parameters])

> The Scan operation returns one or more items and item attributes by accessing every item in a table or a secondary index.

The `scan` method is a wrapper for the DynamoDB Scan API. The DynamoDB Toolbox `scan` method supports all **Scan** API operations. The `scan` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `scanParams` can be used, but will only retrieve the generated parameters.

The `scan()` method accepts two arguments. The first argument is an `options` object that specifies the details of your scan. The following options are all optional (corresponding Scan API references in parentheses):

Option	Type	Description
index	`string`	Name of secondary index to scan. If not specified, the query executes on the primary index. The index must include the table's `entityField` attribute for automatic parsing of returned data. (IndexName)
limit	`number`	The maximum number of items to retrieve per scan. (Limit)
consistent	`boolean`	Enable a consistent read of the items (ConsistentRead)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
select	`string`	The attributes to be returned in the result. One of either `all_attributes`, `all_projected_attributes`, `specific_attributes`, or `count` (Select)

Option	Type	Description
filters	`array` or `object`	A complex `object` or `array` of objects that specifies the scan's filter condition. See Filters and Conditions. (FilterExpression)
attributes	`array` or `object`	An `array` or array of complex `objects` that specify which attributes should be returned. See Projection Expression below (ProjectionExpression)
startKey	`object`	An object that contains the `partitionKey` and `sortKey` of the first item that this operation will evaluate. (ExclusiveStartKey)
segments	`number`	For a parallel `scan` request, `segments` represents the total number of segments into which the `scan` operation will be divided. (TotalSegments)
segment	`number`	For a parallel `scan` request, `segment` identifies an individual segment to be scanned by an application worker. (Segment)
entity	`string`	The name of a table Entity to evaluate `filters` and `attributes` against.
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional second argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

```
let result = await MyTable.scan(
  {
    limit: 100, // limit to 50 items
    capacity: 'indexes', // return the total capacity consumed by the indexes
    filters: { attr: 'total', between: [100,500] }, // only return orders bet
    index: 'GSI1' // scan the GSI1 secondary index
  }
)
```

Return Data

The data is returned with the same response syntax as the DynamoDB Scan API. If `autoExecute` and `autoParse` are enabled, any `Items` data returned will be parsed into its corresponding Entity's aliases. Otherwise, the DocumentClient will return the unmarshalled data. If the response is parsed by the library, a `.next()` method will be available on the returned object. Calling this function will call the `scan` method again using the same parameters and passing the `LastEvaluatedKey` in as the `ExclusiveStartKey`. This is a convenience method for paginating the results.

batchGet(items [,options] [,parameters])

> The BatchGetItem operation returns the attributes of one or more items from one or more tables. You identify requested items by primary key.

The `batchGet` method is a wrapper for the DynamoDB BatchGetItem API. The DynamoDB Toolbox `batchGet` method supports all **BatchGetItem** API operations. The `batchGet` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `batchGetParams` can be used, but will only retrieve the generated parameters.

The `batchGet` method accepts three arguments. The first is an `array` of item keys to get. The DynamoDB Toolbox provides the `getBatch` method on your entities to help you generate the proper key configuration. You can specify different entity types as well as entities from different tables, and this library will handle the proper payload construction.

The optional second argument accepts an `options` object. The following options are all optional (corresponding BatchGetItem API references in parentheses):

Option	Type	Description
consistent	`boolean` or `object` (see below)	Enable a consistent read of the items (ConsistentRead)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)

Option	Type	Description
attributes	`array` or `object` (see below)	An `array` or array of complex `objects` that specify which attributes should be returned. See Projection Expression below (ProjectionExpression)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

Specifying options for multiple tables

The library is built for making working with single table designs easier, but it is possible that you may need to retrieve data from multiple tables within the same batch get. If your `items` contain references to multiple tables, the `consistent` option will accept objects that use either the table `name` or `alias` as the key, and the setting as the value. For example, to specify different `consistent` settings on two tables, you would use something like following:

```
consistent: {
  'my-table-name': true,
  'my-other-table-name': false
}
```

Setting either value without the `object` structure will set the option for all referenced tables. If you are referencing multiple tables and using the `attributes` option, then you must use the same `object` method to specify the table `name` or `alias`. The value should follow the standard Projection Expression formatting.

```
const results = await MyTable.batchGet(
  [
    MyTable.User.getBatch({ family: 'Brady', name: 'Mike' }),
    MyTable.User.getBatch({ family: 'Brady', name: 'Carol' }),
    MyTable.Pet.getBatch({ family: 'Brady', name: 'Tiger' })
  ],
  {
    capacity: 'total',
    attributes: [
        'name', 'family',
        { User: ['dob', 'age'] },
```

```
          { Pet: ['petType','lastVetCheck'] }
        ]
      }
    }
  )
```

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

Return Data

The data is returned with the same response syntax as the DynamoDB BatchGetItem API. If `autoExecute` and `autoParse` are enabled, any `Responses` data returned will be parsed into its corresponding Entity's aliases. Otherwise, the DocumentClient will return the unmarshalled data. If the response is parsed by the library, a `.next()` method will be available on the returned object. Calling this function will call the `batchGet` method again using the same options and passing any `UnprocessedKeys` in as the `RequestItems`. This is a convenience method for retrying unprocessed keys.

batchWrite(items [,options] [,parameters])

> The BatchWriteItem operation puts or deletes multiple items in one or more tables. A single call to BatchWriteItem can write up to 16 MB of data, which can comprise as many as 25 put or delete requests.

The `batchWrite` method is a wrapper for the DynamoDB BatchWriteItem API. The DynamoDB Toolbox `batchWrite` method supports all **BatchWriteItem** API operations. The `batchWrite` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `batchWriteParams` can be used, but will only retrieve the generated parameters.

The `batchWrite` method accepts three arguments. The first is an `array` of item keys to either `put` or `delete`. The DynamoDB Toolbox provides a `putBatch` and `deleteBatch` method on your entities to help you generate the proper key configuration for each item. You can specify different entity types as well as entities from different tables, and this library will handle the proper payload construction.

The optional second argument accepts an `options` object. The following options are all optional (corresponding BatchWriteItem API references in parentheses):

Option	Type	Description

Option	Type	Description
capacity	`string` or `object` (see below)	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
metrics	`string`	Return item collection metrics. If set to `size`, the response includes statistics about item collections, if any, that were modified during the operation are returned in the response. One of either `none` or `size` (ReturnItemCollectionMetrics)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

NOTE: The `BatchWriteItem` does not support conditions or return deleted items. *"BatchWriteItem does not behave in the same way as individual PutItem and DeleteItem calls would. For example, you cannot specify conditions on individual put and delete requests, and BatchWriteItem does not return deleted items in the response."* ~ DynamoDB BatchWriteItem API

```
const result = await Default.batchWrite(
  [
    MyTable.User.putBatch({ family: 'Brady', name: 'Carol', age: 40, roles: [
    MyTable.User.putBatch({ family: 'Brady', name: 'Mike', age: 42, roles: ['
    MyTable.Pet.deleteBatch({ family: 'Brady', name: 'Tiger' })
  ],{
    capacity: 'total',
    metrics: 'size',
  }
)
```

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

Return Data

The data is returned with the same response syntax as the DynamoDB BatchWriteItem API. If `autoExecute` and `autoParse` are enabled, a `.next()` method will be available on the returned object. Calling this function will call the `batchWrite` method again using the same options and passing any `UnprocessedItems` in as the `RequestItems`. This is a convenience method for retrying unprocessed keys.

transactGet(items [,options] [,parameters])

> TransactGetItems is a synchronous operation that atomically retrieves multiple items from one or more tables (but not from indexes) in a single account and Region.

The `transactGet` method is a wrapper for the DynamoDB TransactGetItems API. The DynamoDB Toolbox `transactGet` method supports all **TransactGetItem** API operations. The `transactGet` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `transactGetParams` can be used, but will only retrieve the generated parameters.

The `transacthGet` method accepts three arguments. The first is an `array` of item keys to get. The DynamoDB Toolbox provides the `getTransaction` method on your entities to help you generate the proper key configuration. You can specify different entity types as well as entities from different tables, and this library will handle the proper payload construction.

The optional second argument accepts an `options` object. The following options are all optional (corresponding TransactGetItems API references in parentheses):

Option	Type	Description
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Table*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Table*)

Accessing items from multiple tables

Transaction items are atomic, so each `Get` contains the table name and key necessary to retrieve the item. The library will automatically handle adding the necessary information and will parse each entity automatically for you.

```
const results = await MyTable.transactGet(
  [
    User.getTransaction({ family: 'Brady', name: 'Mike' }),
    User.getTransaction({ family: 'Brady', name: 'Carol' }),
    Pet.getTransaction({ family: 'Brady', name: 'Tiger' })
  ],
  { capacity: 'total' }
)
```

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

Return Data

The data is returned with the same response syntax as the DynamoDB TransactGetItems API. If `autoExecute` and `autoParse` are enabled, any `Responses` data returned will be parsed into its corresponding Entity's aliases. Otherwise, the DocumentClient will return the unmarshalled data.

transactWrite(items [,options] [,parameters])

> TransactWriteItems is a synchronous write operation that groups up to 25 action requests. The actions are completed atomically so that either all of them succeed, or all of them fail.

The `transactWrite` method is a wrapper for the DynamoDB TransactWriteItems API. The DynamoDB Toolbox `transactWrite` method supports all **TransactWriteItems** API operations. The `transactWrite` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `transactWriteParams` can be used, but will only retrieve the generated parameters.

The `transactWrite` method accepts three arguments. The first is an `array` of item keys to either `put`, `delete`, `update` or `conditionCheck`. The DynamoDB Toolbox provides `putTransaction`, `deleteTransaction`, `updateTransaction`, and `conditionCheck` methods on your entities to help you generate the proper configuration for each item. You can specify different entity types as well as entities from different tables, and this library will handle the proper payload construction.

The optional second argument accepts an `options` object. The following options are all optional (corresponding TransactWriteItems API references in parentheses):

Option	Type	Description
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
metrics	`string`	Return item collection metrics. If set to `size`, the response includes statistics about item collections, if any, that were modified during the operation are returned in the response. One of either `none` or `size` (ReturnItemCollectionMetrics)
token	`string`	Optional token to make the call idempotent, meaning that multiple identical calls have the same effect as one single call. (ClientRequestToken)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

```
const result = await Default.transactWrite(
  [
    Pet.conditionCheck({ family: 'Brady', name: 'Tiger' }, { conditions: { at
    Pet.deleteTransaction({ family: 'Brady', name: 'Tiger' }),
    User.putTransaction({ family: 'Brady', name: 'Carol', age: 40, roles: ['m
    User.putTransaction({ family: 'Brady', name: 'Mike', age: 42, roles: ['fa
  ],{
    capacity: 'total',
    metrics: 'size',
  }
)
```

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

Return Data

The data is returned with the same response syntax as the DynamoDB TransactWriteItems API.

parse(entity, input [,include])

Executes the `parse` method of the supplied `entity`. The `entity` must be a `string` that references the name of an Entity associated with the table. See the Entity `parse` method for additional parameters and behavior.

get(entity, key [,options] [,parameters])

Executes the `get` method of the supplied `entity`. The `entity` must be a `string` that references the name of an Entity associated with the table. See the Entity `get` method for additional parameters and behavior.

delete(entity, key [,options] [,parameters])

Executes the `delete` method of the supplied `entity`. The `entity` must be a `string` that references the name of an Entity associated with the table. See the Entity `delete` method for additional parameters and behavior.

put(entity, item [,options] [,parameters])

Executes the `put` method of the supplied `entity`. The `entity` must be a `string` that references the name of an Entity associated with the table. See the Entity `put` method for additional parameters and behavior.

update(entity, key [,options] [,parameters])

Executes the `update` method of the supplied `entity`. The `entity` must be a `string` that references the name of an Entity associated with the table. See the Entity `update` method for additional parameters and behavior.

Entity Properties

get/set `table`

Retrieves a reference to the Table instance that the Entity is attached to. You can use this property to add the Entity to a Table by assigning it a valid Table instance. Note that you cannot change a table once it has been assigned.

get `DocumentClient`

The `DocumentClient` property retrieves a reference to the table's assigned `DocumentClient`. This value cannot be updated by the Entity.

get/set `autoExecute`

This property will retrieve a `boolean` indicating the current `autoExecute` setting on the entity. If no value is set, it will return the inherited value from the attached table. You can change this setting for the current entity by supplying a `boolean` value.

get/set `autoParse`

This property will retrieve a `boolean` indicating the current `autoParse` setting on the entity. If no value is set, it will return the inherited value from the attached table. You can change this setting for the current entity by supplying a `boolean` value.

get `partitionKey`

Returns the Entity's assigned `partitionKey`.

get `sortKey`

Returns the Entity's assigned `sortKey`.

Entity Methods

attribute(attribute)

Returns the Table's attribute name for the suppled `attribute`. The `attribute` must be a `string` and can be either a valid attribute name or alias.

parse(input [,include])

Parses attributes returned from a DynamoDB action and unmarshalls them into entity aliases. The `input` argument accepts an `object` with attributes as keys, an `array` of `objects` with attributes as keys, or an `object` with either an `Item` or `Items` property. This method will return a result of the same type of `input`. For example, if you supply an `array` of objects, an `array` will be returned. If you supply an object with an `Item` property, an `object` will be returned.

You can also pass in an `array` of strings as the second argument. The unmarshalling will only return the attributes (or aliases) specified in this `include` array.

If auto execute and auto parsing are enable, data returned from a DynamoDB action will automatically be parsed.

get(key [,options] [,parameters])

> The GetItem operation returns a set of attributes for the item with the given primary key.

The `get` method is a wrapper for the DynamoDB GetItem API. The DynamoDB Toolbox `get` method supports all **GetItem** API operations. The `get` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `getParams` can be used, but will only retrieve the generated parameters.

The `get` method accepts three arguments. The first argument accepts an `object` that is used to specify the primary key of the item you wish to "get" (Key). The `object` must contain keys for the attributes that represent your `partitionKey` and `sortKey` (if a compound key) with their values as the key values. For example, if `user_id` represents your `partitionKey`, and `status` represents your `sortKey`, to retrieve user_id "123" with a status of "active", you would specify `{ user_id: 123, status: 'active' }` as your `key`.

The optional second argument accepts an `options` object. The following options are all optional (corresponding GetItem API references in parentheses):

Option	Type	Description
consistent	`boolean`	Enable a consistent read of the items (ConsistentRead)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
attributes	`array` or `object`	An `array` or array of complex `objects` that specify which attributes should be returned. See Projection Expression below (ProjectionExpression)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

```
// Specify my key
let key = {
  id: 123,
  status: 'active',
  date_added: '2020-04-24'
}

// Use the 'get' method of MyEntity to retrieve the item from DynamoDB
let result = await MyEntity.get(
  key,
  { consistent: true }
)
```

delete(key [,options] [,parameters])

> Deletes a single item in a table by primary key.

The `delete` method is a wrapper for the DynamoDB DeleteItem API. The DynamoDB Toolbox `delete` method supports all **DeleteItem** API operations. The `delete` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `deleteParams` can be used, but will only retrieve the generated parameters.

The `delete` method accepts three arguments. The first argument accepts an `object` that is used to specify the primary key of the item you wish to "delete" (Key). For example: `{ user_id: 123, status: 'active' }`

The optional second argument accepts an `options` object. The following options are all optional (corresponding DeleteItem API references in parentheses):

Option	Type	Description
conditions	`array` or `object`	A complex `object` or `array` of objects that specifies the conditions that must be met to delete the item. See Filters and Conditions. (ConditionExpression)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)

Option	Type	Description
metrics	`string`	Return item collection metrics. If set to `size`, the response includes statistics about item collections, if any, that were modified during the operation are returned in the response. One of either `none` or `size` (ReturnItemCollectionMetrics)
returnValues	`string`	Determins whether to return item attributes as they appeared before they were deleted. One of either `none` or `all_old`. (ReturnValues)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

```
// Specify my key
let key = {
  id: 123,
  status: 'active',
  date_added: '2020-04-24'
}

// Use the 'delete' method of MyEntity to delete the item from DynamoDB
let result = await MyEntity.delete(
  key,
  condition: { attr: 'date_modified' lt: '2020-01-01' },
  returnValues: 'all_old'
)
```

put(item [,options] [,parameters])

> Creates a new item, or replaces an old item with a new item. If an item that has the same primary key as the new item already exists in the specified table, the new item completely replaces the existing item.

The `put` method is a wrapper for the DynamoDB PutItem API. The DynamoDB Toolbox `put` method supports all **PutItem** API operations. The `put` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `putParams` can be used, but will only retrieve the generated parameters.

The `put` method accepts three arguments. The first argument accepts an `object` that represents the item to add to the DynamoDB table. The item can use attribute names or aliases and will convert the object into the appropriate shape defined by your Entity.

The optional second argument accepts an `options` object. The following options are all optional (corresponding PutItem API references in parentheses):

Option	Type	Description
conditions	`array` or `object`	A complex `object` or `array` of objects that specifies the conditions that must be met to put the item. See Filters and Conditions. (ConditionExpression)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)
metrics	`string`	Return item collection metrics. If set to `size`, the response includes statistics about item collections, if any, that were modified during the operation are returned in the response. One of either `none` or `size` (ReturnItemCollectionMetrics)
returnValues	`string`	Determins whether to return item attributes as they appeared before a new item was added. One of either `none` or `all_old`. (ReturnValues)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters. See Adding custom parameters and clauses for more information.

```
// Create my item (using table attribute names or aliases)
let item = {
  id: 123,
  name: 'Jane Smith',
  company: 'ACME',
  age: 35,
  status: 'active',
  date_added: '2020-04-24'
}

// Use the 'put' method of your entity instance
let result = await MyEntity.put(item)
```

update(key [,options] [,parameters])

> Edits an existing item's attributes, or adds a new item to the table if it does not already exist. You can put, delete, or add attribute values.

The `update` method is a wrapper for the DynamoDB UpdateItem API. The DynamoDB Toolbox `update` method supports all **UpdateItem** API operations. The `update` method returns a `Promise` and you must use `await` or `.then()` to retrieve the results. An alternative, synchronous method named `updateParams` can be used, but will only retrieve the generated parameters.

The `update` method accepts three arguments. The first argument accepts an `object` that represents the item key and attributes to be updated. The item can use attribute names or aliases and will convert the object into the appropriate shape defined by your Entity.

The optional second argument accepts an `options` object. The following options are all optional (corresponding UpdateItem API references in parentheses):

Option	Type	Description
conditions	`array` or `object`	A complex `object` or `array` of objects that specifies the conditions that must be met to update the item. See Filters and Conditions. (ConditionExpression)
capacity	`string`	Return the amount of consumed capacity. One of either `none`, `total`, or `indexes` (ReturnConsumedCapacity)

Option	Type	Description
metrics	`string`	Return item collection metrics. If set to `size`, the response includes statistics about item collections, if any, that were modified during the operation are returned in the response. One of either `none` or `size` (ReturnItemCollectionMetrics)
returnValues	`string`	Determins whether to return item attributes as they appeared before or after the item was updated. One of either `none`, `all_old`, `updated_old`, `all_new`, `updated_new`. (ReturnValues)
execute	`boolean`	Enables/disables automatic execution of the DocumentClient method (default: *inherited from Entity*)
parse	`boolean`	Enables/disables automatic parsing of returned data when `autoExecute` evaluates to `true` (default: *inherited from Entity*)

If you prefer to specify your own parameters, the optional third argument allows you to add custom parameters and clauses. See Adding custom parameters and clauses for more information.

But wait, there's more! The `UpdateExpression` lets you do all kinds of crazy things like `REMOVE` attributes, `ADD` values to numbers and sets, and manipulate arrays. The DynamoDB Toolbox has simple ways to deal with all these different operations by properly formatting your input data.

Updating an attribute

To update an attribute, include the key and any fields that you want to update.

```
let item = {
   id: 123,
   sk: 'abc',
   status: 'inactive',
}
await MyEntity.update(item)
```

Removing an attribute

To remove attributes, add a `$remove` key to your item and provide an array of attributes or aliases to remove.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  $remove: ['roles','age']
}
```

Adding a number to a `number` attribute

DynamoDB lets us add (or subtract) numeric values from an attribute in the table. If no value exists, it simply puts the value. Adding with the DynamoDB Toolbox is just a matter of supplying an `object` with an `$add` key on the number fields you want to update.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  level: { $add: 2 } // add 2 to level
}
```

Adding values to a `set`

Sets are similar to lists, but they enforce unique values of the same type. To add new values to a set, use an `object` with an `$add` key and an array of values.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  roles: { $add: ['author','support'] }
}
```

Deleting values from a `set`

To delete values from a `set`, use an `object` with a `$delete` key and an array of values to delete.

```
let item = {
  id: 123,
  name: 'Test Name',
```

```
  status: 'active',
  date_added: '2020-04-24',
  roles: { $delete: ['admin'] }
}
```

Appending (or prepending) values to a `list`

To append values to a `list`, use an `object` with an `$append` key and an array of values to append.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  sessions: { $append: [ { date: '2020-04-24', duration: 101 } ] }
}
```

Alternatively, you can use the `$prepend` key and it will add the values to the beginning of the list.

Remove items from a `list`

To remove values from a `list`, use an `object` with a `$remove` key and an array of **indexes** to remove. Lists are indexed starting at `0`, so the update below would remove the second, fifth, and sixth item in the array.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  sessions: { $remove: [1,4,5] }
}
```

Update items in a `list`

To update values in a `list`, specify an `object` with array indexes as the keys and the update data as the values. Lists are indexed starting at `0`, so the update below would update the second and fourth items in the array.

```
let item = {
  id: 123,
```

```
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  sessions: {
    1: 'some new value for the second item',
    3: 'new value for the fourth value'
  }
}
```

Update nested data in a `map`

Maps can be complex, deeply nested JavaScript objects with a variety of data types. The DynamoDB Toolbox doesn't support schemas for `map` s (yet), but you can still manipulate them by wrapping your updates in a `$set` parameter and using dot notation and array index notation to target fields.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  metadata: {
    $set: {
      'title': 'Developer', // update metadata.title
      'contact.name': 'Jane Smith', // update metadata.contact.name
      'contact.addresses[0]': '123 Main Street' // update the first array ite
    }
  }
}
```

We can also use our handy `$add` , `$append` , `$prepend` , and `$remove` properties to manipulate nested values.

```
let item = {
  id: 123,
  name: 'Test Name',
  status: 'active',
  date_added: '2020-04-24',
  metadata: {
    $set: {
      'vacation_days': { $add: -2 },
      'contact.addresses': { $append: ['99 South Street'] },
      'contact.phone': { $remove: [1,3] }
    }
  }
}
```

query(partitionKey [,options] [,parameters])

Executes the `query` method on the parent Table. This method accepts the same parameters as the Table `query` method and automatically sets the `entity` option to the current entity. Due to the nature of DynamoDB queries, this method **does not** guarantee that only items of the current entity type will be returned.

scan([options] [,parameters])

Executes the `scan` method on the parent Table. This method accepts the same parameters as the Table `scan` method and automatically sets the `entity` option to the current entity. Due to the nature of DynamoDB scans, this method **does not** guarantee that only items of the current entity type will be returned.

Filters and Conditions

DynamoDB supports **Filter** and **Condition** expressions. **Filter Expressions** are used to limit data returned by `query` and `scan` operations. **Condition Expressions** are used for data manipulation operations (`put` , `update` , `delete` and `batchWrite`), allowing you to specify a condition to determine which items should be modified.

The DynamoDB Toolbox provides an **Expression Builder** that allows you to generate complex filters and conditions based on your Entity definitions. Any method that requires `filters` or `conditions` accepts an `array` of *conditions*, or a single *condition*. *Condition* objects support the following properties:

Properties	Type	Description
attr	`string`	Specifies the attribute to filter on. If an `entity` property is provided (or inherited from the calling operation), aliases can be used. Either `attr` or `size` must be provided.
size	`string`	Specifies which attribute's calculated size to filter on (see Operators and Functions for more information). If an `entity` property is provided (or inherited from the calling operation), aliases can be used. Either `attr` or `size` must be provided.
eq	*	Specifies value to *equal* attribute or size of attribute.

Properties	Type	Description
ne	*	Specifies value to *not equal* attribute or size of attribute.
lt	*	Specifies value for attribute or size to be *less than*.
lte	*	Specifies value for attribute or size to be *less than or equal to*.
gt	*	Specifies value for attribute or size to be *greater than*.
gte	*	Specifies value for attribute or size to be *greater than or equal to*.
between	`array`	Specifies values for attribute or size to be *between*. E.g. `[18,49]`.
beginsWith	*	Specifies value for the attribute to *begin with*
in	`array`	Specifies and `array` of values that the attribute or size must match one value.
contains	`string`	Specifies value that must be contained within a string or Set. (see Operators and Functions for more information)
exists	`boolean`	Checks whether or not the attribute exists for an item. A value of `true` uses the `attribute_exists()` function and a value of `false` uses the `attribute_not_exists()` function (see Operators and Functions for more information)
type	`string`	A value that compares the attribute's type. Value must be one of `S`, `SS`, `N`, `NS`, `B`, `BS`, `BOOL`, `NULL`, `L`, or `M` (see Operators and Functions for more information)
or	`boolean`	Changes the logical evaluation to `OR` (by default it's `AND`)
negate	`boolean`	Adds `NOT` to the condition.
entity	`string`	The entity this attribute applies to. If supplied (or inherited from the calling operation), `attr` and `size` properties can use the entity's aliases to reference attributes.

* Comparison values should equal the type of the attribute you are comparing against. If you are using the `size` property, the value should be a `number`.

Complex Filters and Conditions

In order to create complex filters and conditions, the DynamoDB Toolbox allows you to nest and combine filters by using nested `array`s. Array brackets (`[` and `]`) act as parentheses when constructing your condition. Using `or` in the first condition within an array will change the logical evaluation for group of conditions.

Condition where `age` is between 18 and 54 **AND** `region` equals "US":

```
filters: [
  { attr: 'age', between: [18,54]},
  { attr: 'region', eq: 'US' }
]
```

Condition where `age` is between 18 and 54 **AND** `region` equals "US" **OR** "EU":

```
filters: [
  { attr: 'age', between: [18,54]},
  [
    { attr: 'region', eq: 'US' },
    { or: true, attr: 'region', eq: 'EU' }
  ]
]
```

Condition where `age` is greater than 21 **OR** ((`region` equals "US" **AND** `interests` size is greater than 10) **AND** `interests` contain `nodejs` , `dynamodb` , or `serverless`):

```
filters: [
  { attr: 'age', gt: 21},
  [
    [
      { or: true, attr: 'region', eq: 'US' },
      { size: 'interests', gt: 10 }
    ],
    [
      { attr: 'interests', contains: 'nodejs' }
      { or: true, attr: 'interests', contains: 'dynamodb' }
      { or: true, attr: 'interests', contains: 'serverless' }
    ]
  ]
]
```

Projection Expressions

DynamoDB supports Projection Expressions that allow you to selectively return attributes when using the `get`, `query` or `scan` operations.

The DynamoDB Toolbox provides a **Projection Builder** that allows you to generate `ProjectionExpression`s that automatically generates `ExpressionAttributeNames` as placeholders to avoid reservered word collisions. The library allows you to work with both table attribute names and Entity aliases to specify projections.

Read operations that provide an `attributes` property accept an `array` of attribute names and/or objects that specify the Entity as the key with an array of attributes and aliases.

Retrieve the `pk`, `sk`, `name` and `created` attributes for all items:

```
attributes: [ 'pk', 'sk', 'name', 'created' ]
```

Retrieve the `user_id`, `status`, and `created` attributes for the `User` entity:

```
attributes: [{ User: [ 'user_id', 'status', 'created' ] }]
```

Retrieve the `pk`, `sk`, and `type` attributes for all items, the `user_id` for the `User` entity, and the `status` and `created` attributes for the the `Order` entity:

```
attributes: [
  'pk', 'sk', 'type',
  { User: [ 'user_id' ] },
  { Order: [ 'status', 'created' ] }
]
```

When using the `get` method of an entity, the "entity" is assumed for the attributes. This lets you specify attributes and aliases without needing to use the object reference.

NOTE: When specifying entities in `query` and `scan` operations, it's possible that shared attributes will retrieve data for other matching entity types. However, the library attempts to return only the attributes specified for each entity when parsing the response.

Adding Custom Parameters and Clauses

This libary supports all API options for the available API methods, so it is unnecessary for you to provide additional parameters. However, if you would like to pass custom parameters, simply pass them in an object as the last parameter to any appropriate method.

```
let result = await MyEntity.update(
  item, // the item to update
  { ..options... }, // method options
  { // your custom parameters
    ReturnConsumedCapacity: 'TOTAL',
    ReturnValues: 'ALL_NEW'
  }
)
```

For the `update` method, you can add additional statements to the clauses by specifying arrays as the `SET`, `ADD`, `REMOVE` and `DELETE` properties. You can also specify additional `ExpressionAttributeNames` and `ExpressionAttributeValues` with object values and the system will merge them in with the generated ones.

```
let results = await MyEntity.update(item, {}, {
  SET: ['#somefield = :somevalue'],
  ExpressionAttributeNames: { '#somefield': 'somefield' },
  ExpressionAttributeValues: { ':somevalue': 123 }
})
```

Additional References

- DocumentClient SDK Reference
- Best Practices for DynamoDB
- DynamoDB, explained.
- The DynamoDB Book

Sponsors

Contributions and Feedback

Contributions, ideas and bug reports are welcome and greatly appreciated. Please add issues for suggestions and bug reports or create a pull request. You can also contact me on Twitter: @jeremy_daly.

Releases 2

v0.3.0 Latest
7 days ago

+ 1 release

Packages

No packages published

Used by 38

+ 30

Contributors 10

Languages

TypeScript 99.9% JavaScript 0.1%