**simple-dynamo**

is a abstraction layer to Jed Schmidt's [dynamo](https://github.com/jed/dynamo) Node.js driver.

[Build Status](http://travis-ci.org/mpneuried/simple-dynamo) *Travis currently fails because the credentials are no longer vaild*

It provides a absolute simple JSON-CRUD Interface without any knowledge of Dynamos specialties.

A special feature is the *combineTableTo* options for tables. It adds the ability to combine multiple models into one Dymamo-table, but use them separately in your application. So you have to pay only one throughput capacity.

**Need a offline version of Dynamo?** [mysql-dynamo](http://mpneuried.github.io/mysql-dynamo/) is a solution to use the same interface of this simple-dynamo module with a MySQL database.

*Written in coffee-script*

**INFO: all examples are written in coffee-script**

**Install**

npm install simple-dynamo

**Connection and Table**

**Initialize**

first you have to define the connection and table attributes and get an instance of the simple-dynamo interface.

new SimpleDynamo( connectionSettings, tables )

###connection Settings

* **accessKeyId** : Your AWS access key id
* **secretAccessKey** : Your AWS secret access key
* **region** : The region your Dynamo-Tables will be placed
* **tablePrefix** : A general table prefix which will be added before all real dynamo table names

###table Definition

An Object of Tables.  
The key you are using will be the key to get the table object.

Every object can have the following keys:

* **name** : *( String required )*  
  Tablename for AWS
* **hashKey** : *( String required )*  
  The hash key name of your ids/hashes
* **hashKeyType** : *( String optional: default = S )*  
  The type of the hashKey. Possible values are: S = String and N = Numeric
* **rangeKey**: *( String optional )*  
  The range key name of your range attribute. If not defined the table will be generated without the range methods
* **rangeKeyType**: *( String optional: default = N )*  
  The type of the rangeKey. Possible values are: S = String and N = Numeric
* **fnCreateHash**: *( Function optional: default = new UUID )*  
  Method to generate a custom hash key.
* **defaultfields**: *( Array optional )*  
  List of fields to return as default. If nothing is defined all fields will be received. You always can overwrite this using options.fields.
* **combineTableTo**: *( String optional )* Option to combine multiple models into one dynamo-table. Makes sense if you want to pay only one table. Combinations are not allowed for tables of different types ( Hash-table and HashRange-table ) and you have to use the same hashKey and rangeKey. The module will handle all interactions with the models transparent, so you only have to define this option.  
  *Note:* If you use this feature and predefine the id/hash you have to add the name of the table in front of every id/hash.
* **overwriteExistingHash**: *( Boolean optional: default = false )*  
  Overwrite a item on create of an existing hash.
* **consistent**: *( Boolean optional: default = false )*  
  Do consistend reads on table.get() and table.find() as default  
  **Method Arguments**
  + **attributes**: The given attributes on create
  + **cb**: Callback method to pass the custom generates id/hash. cb( "my-special-hash" )
* **fnCreateRange**: *( Function optional: default = current Timestamp )*  
  Method to generate a custom range key.  
  **Method Arguments**
  + **attributes**: The given attributes on create
  + **cb**: Callback method to pass the custom generates id/hash. cb( "my-special-range" )
* **attributes**: *( Array of Objects required )*  
  An array of attribute Objects. Which will be validated  
  **Attributes keys**
  + **key**: *( String required )*  
    Column/Attribute name/key
  + **type**: *( String required )*  
    Datatype. possible values are string = String, number = Numeric and array = Array/Set of **Strings**
  + **required**: *( Boolean optional: default = false )*  
    Validate the attribute to be required. *( Not implemented yet ! )*

**Example**

# import module

SimpleDynamo = require "simple-dynamo"

# define connection settings

connectionSettings =

accessKeyId: "-"

secretAccessKey: "-"

region: "eu-west-1"

tablePrefix: null

# define tables

tables =

"Users":

name: "users"

hashKey: "id"

attributes: [

{ key: "name", type: "string", required: true }

{ key: "email", type: "string" }

]

"Todos":

name: "todos"

hashKey: "id"

rangeKey: "\_t"

rangeKeyType: "N"

fnCreateHash: ( attributes, cb )=>

cb( attributes.user\_id )

return

attributes: [

{ key: "title", type: "string", required: true }

{ key: "done", type: "number" }

]

# example for a combined table usage

"Combined1":

name: "c1"

hashKey: "id"

combineTableTo: "combined\_hash"

attributes: [

{ key: "title", type: "string", required: true }

]

"Combined2":

name: "c2"

hashKey: "id"

combineTableTo: "combined\_hash"

attributes: [

{ key: "title", type: "string", required: true }

]

# create instance

sdManager = new SimpleDynamo( connectionSettings, tables )

# connect

sdManager.connect ( err )->

console.log( "simple-dynamo ready to use" )

**First connect to AWS**

The module has to know about the existing AWS tables so you have to read them first.  
**If you do not run .connect() the module will throw an error everytime**

**Manager.connect( fnCallback ) Arguments** :

* **fnCallback**: *( Function required )*  
  Callback method. Single arguments on return is the error object. On success the error is null

**Example**

sdManager.connect ( err )->

if err

console.error( "connect ERROR", err )

else

console.log( "simple-dynamo ready to use" )

**Create all tables**

to create all missing tables just call .createAll().

This is not necessary if you know the tables has been created in the past.

**Note! The generating of tables could take a few Minutes**

**Manager.generateAll( fnCallback ) Arguments** :

* **fnCallback**: *( Function required )*  
  Callback method. Single arguments on return is the error object. On success the error is null

**Example**

sdManager.generateAll ( err )->

if err

console.error( "connect ERROR", err )

else

console.log( "simple-dynamo ready to use" )

**Get a table instance**

To interact with a table you have to retrieve the table object. It's defined in the table-definitions

**Manager.get( 'tableName' ) Arguments** :

* **tableName**: *( String required )*  
  Method to retrieve the instance of a table object.

**Example**

tblTodos = sdManager.get( 'Todos' )

**Destroy a table**

destroy table at AWS. This removes the table from AWS will all the data

**Table.destroy( fnCallback ) Arguments** :

* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg

**Example**

tblTodos.del ( err )->

if err

console.error( "destroy ERROR", err )

else

console.log( "table destroyed" )

**Item handling**

**Write a new item (INSERT)**

Create a new item in a select table. You can also add some attributes not defined in the table-definition, which will be saved, too.

**Table.set( data, options, fnCallback ) Arguments** :

* **data**: *( Object required )*  
  The data to save. You can define the hash and/or range key. If not the module will generate a hash/range automatically.  
  *Note:* If the used table uses the combined feature and you define the hash-key it's necessary to add the name out of the table-config in front of every hash.
* **options**: *( Object optional )*
  + **fields**: *( Array )* An array of fields to receive
  + **overwriteExistingHash**: *( Boolean optional: default = [tableConfig.overwriteExistingHash] )* Overwrite a item it already exists.
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg
  + **item**: the save item as simple object

**Example**

data =

title: "My First Todo"

done: 0

aditionalData: "Foo bar"

tblTodos.set data, ( err, todo )->

if err

console.error( "insert ERROR", err )

else

console.log( todo )

**Get a item (GET)**

Get an existing element by id/hash

**Table.get( id, fnCallback ) Arguments** :

* **id**: *( String|Number|Array required )*  
  The id of an element. If the used table is a range table you have to use an array [hash,range] as combined id. Otherwise you will get an error.
* **options**: *( Object optional )*
  + **fields**: *( Array )* An array of fields to receive. If nothing is defined all fields are returned.
  + **consistent**: *( Boolean optional: default = [tableConfig.consistent] )* do a consitent read
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg
  + **item**: the database item as simple object. If not found null

**Example**

tblTodos.get 'myTodoId', ( err, todo )->

if err

console.error( "get ERROR", err )

else

console.log( todo )

tblRangeTodos.get [ 'myHash', 'myRange' ], ( err, todo )->

if err

console.error( "get ERROR", err )

else

console.log( todo )

**Get many items (MGET)**

Get an many existing elements by id/hash in one request

**Table.mget( [ id1, id2, .. ], options, fnCallback ) Arguments** :

* **ids**: *( Array required )*  
  An array of id of an elements. If the used table is a range table you have to use an array of arrays [hash,range] as combined id. Otherwise you will get an error.
* **options**: *( Object optional )*
  + **fields**: *( Array )* An array of fields to receive. If nothing is defined all fields are returned.
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg
  + **items**: the database items as a array of simple objects. Only existing items will be received.

**Example**

tblTodos.mget [ 'myTodoIdA', 'myTodoIdB' ], ( err, todos )->

if err

console.error( "get ERROR", err )

else

console.log( todos )

tblRangeTodos.mget [ [ 'myHash', 1 ], [ 'myHash', 2 ] ], ( err, todos )->

if err

console.error( "get ERROR", err )

else

console.log( todos )

**Update an item (UPDATE)**

update an existing item.  
To remove a attribute you have to set the value to null

**Table.set( id, data, options, fnCallback ) Arguments** :

* **id**: *( String|Number|Array required )*  
  The id of an element. If the used table is a range table you have to use an array [hash,range] as combined id. Otherwise you will get an error.
* **data**: *( Object required )*  
  The data to update. You can redefine the range key. If you pass the hash key it will be ignored
* **options**: *( Object optional )*  
  For update you can define some options.
  + **fields**: *( Array )* An array of fields to receive
  + **conditionals** *( Object )* A query object to define a conditional. Only {"==": value}, {"==": null}, and {"!=": null} are allowed. How to build? … have a look at [Jed's Predicates](https://github.com/jed/dynamo/wiki/High-level-API#wiki-predicates)
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg
  + **item**: the database item as simple object. If not found null

**Example**

data =

title: "My First Update"

done: 1

tblTodos.set 'myTodoId', data, ( err, todo )->

if err

console.error( "update ERROR", err )

else

# note. the key 'aditionalData' will be gone

console.log( todo )

**Delete an item (DELETE)**

delete an item by id/hash

**Table.del( id, fnCallback ) Arguments** :

* **id**: *( String|Number|Array required )*  
  The id of an element. If the used table is a range table you have to use an array [hash,range] as combined id. Otherwise you will get an error.
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg

**Example**

tblTodos.del 'myTodoId', ( err )->

if err

console.error( "delete ERROR", err )

else

console.log( "delete done" )

**Query a table (FIND)**

run a query on a table. The module automatically trys to do a Dynamo.db scan or Dynamo query.

**Table.find( query, startAt, options, fnCallback ) Arguments** :

* **query**: *( Object : default = {} all )*  
  A query object. How to build … have a look at [Jed's Predicates](https://github.com/jed/dynamo/wiki/High-level-API#wiki-predicates)
* **startAt**: *( String|Number|Array optional )*  
  To realize a paging you can define a startAt. Usually the last item of a list. If you define startAt with the last item of the previous find you get the next collection of items without the given startAt item.  
  If the used table is a range table you have to use an array [hash,range] as combined startAt. Otherwise you will get an error.
* **options**: *( Object optional )*
  + **fields**: *( Array )* An array of fields to receive
  + **limit**: *( Number )* Define the max. items to return
  + **forward**: *( Boolean default = true )* define the direction acs or desc for range querys.
  + **consistent**: *( Boolean optional: default = [tableConfig.consistent] )* do a consitent read.  
    **Note! This is only availible for real query's through range-tables. This means if you query by hash and range ( e.g. Advanced Examples )**
* **fnCallback**: *( Function required )*  
  Callback method.  
  **Method Arguments**
  + **err**: Usually null. On an error a object with error and msg
  + **items**: an array of objects found

**Example**

tblTodos.find {}, ( err, items )->

if err

console.error( "delete ERROR", err )

else

console.log( "all existend items", items )

**Advanced Examples**

# create a query to read all todos from last hour

\_query =

id: { "!=": null }

\_t: { "<": ( Date.now() - ( 1000 \* 60 \* 60 ) ) }

tblTodos.find , ( err, items )->

if err

console.error( "delete ERROR", err )

else

console.log( "found items", items )

# read 4 todos from last hour beginning starting with a known id

\_query =

id: { "!=": null }

\_t: { "<": ( Date.now() - ( 1000 \* 60 \* 60 ) ) }

\_startAt = "myid\_todoItem12"

\_options = { "limit": 4, "fields": [ "id", "\_t", "title" ] }

tblTodos.find \_query, \_startAt, \_options, ( err, items )->

if err

console.error( "delete ERROR", err )

else

console.log( "4 found items", items )

**Working with sets**

Dynamo has the ability to work with sets. That means you can save a Set of Strings as an Array.  
During an update you have the ability to add or remove a single value out of the set. Or you can reset the whole set.

But you can only perform one action per key and you obnly can use the functionalty if defined through the table-definition ( type:"array" ).

Existing values will be ignored.

The following key variants are availible:

* "key":[ "a", "b", "c" ]': Resets the whole value of the key
* "key":{ "$add": [ "d", "e" ] }: Add some values to the set
* "key":{ "$rem": [ "a", "b" ] }: remove some values
* "key":{ "$reset": [ "x", "y" ] }: reset the whole value. Same as "key":[ "x", "y" ]'
* "key":{ "$add": "d"}: Add a single value to the set
* "key":{ "$rem": "a" }: remove a single value
* "key":{ "$reset": "y" }: reset the whole set to a single value. Same as "key":[ "y" ]'

**Examples**

# Source "key: [ "a", "b", "c" ]"

data =

key: [ "x", "y", "z" ]

tblSets.set 'mySetsId', data, ( err, setData )->

# Result "key: [ "x", "y", "z" ]"

console.log( setData )

# Source "key: [ "a", "b", "c" ]"

data =

key: { "$add": [ "a", "d", "e" ] }

tblSets.set 'mySetsId', data, ( err, setData )->

# Result "key: [ "a", "b", "c", "d", "e" ]"

console.log( setData )

# Source "key: [ "a", "b", "c" ]"

data =

key: { "$rem": [ "a", "b", "x" ] }

tblSets.set 'mySetsId', data, ( err, setData )->

# Result "key: [ "c" ]"

console.log( setData )

# Source "key: [ "a", "b", "c" ]"

data =

key: { "$reset": [ "x", "y", "z" ] }

tblSets.set 'mySetsId', data, ( err, setData )->

# Result "key: [ "x", "y", "z" ]"

console.log( setData )

#Events

To provide a API to react on different events you can listen to a bunch of events.

##Manager Events

* new-table: Table object initialized and ready to use. This means only the client model is ready. Eventually you have to create the table first.  
  **Event Arguments**
  + **name**: the name og the table, like you would use with `Manager.get()
  + **Table**: the Table object
* table-generated: Fired after all a new tables has been generated. **Event Arguments**
  + **Meta**: the tables meta-data
* all-tables-generated: Fired after all tables are generated.

##Table Events

* create-status: fired on table create.  
  **Event Arguments**
  + **status**: describes the state of table creation. Possible values are: already-active, waiting, active
* get: fired after a table.get.  
  **Event Arguments**
  + **item**: the item
* get-empty: fired after a table.get with an empty result.
* mget: fired after a table.mget.  
  **Event Arguments**
  + **items**: the items
* mget-empty: fired after a table.mget with no results.
* create: fired after a item has been created.  
  **Event Arguments**
  + **item**: the created item
* update: fired after a item has been updated.  
  **Event Arguments**
  + **item\_new**: the item after the update
* delete: fired after a item has been deleted.  
  **Event Arguments**
  + **item\_old**: the item before the delete

**Changelogs**

**0.6.2**

* fixed handling ob map/opbject types

**0.6.1**

* fixed parsing of boolean values

**0.6.0**

* updated mp-dynamo to version 1.0.0 so region eu-central-1 with signature version 4 should be possible

**0.5.0**

* fixed create table
* smaller bug fixes due to switch to newer api version by 0.4.x
* optimized published files and dev env
* fixed tests

**0.4.3**

* fixed usage of fields= all for mget + recompile with newer coffe version

**0.4.2**

* fixed usage of new npm module mp-dynamo

**0.4.1**

* bugfix in mp-dynamo that is now a new npm module to load the dependency via npm and get rid of the github dep.

**0.3.12**

* fixed git link to use the old dynamo api version

**0.4.0**

* added newer dynamo datatypes: BOOL, NULL, M, L, B

**0.3.11**

* correct handling of a string predicate with filter for null

**0.3.10**

* toString radix bugfix

**0.3.9**

* removed log output

**0.3.8**

* Added option to be able to increment a numeric by n. To use this you can set the field e.g. to table\_key: { "$add": 1 }

**0.3.7**

* Bugfix for correct handling of BETWEEN predicates

**0.3.6**

* Added conditionals to delete

**0.3.5**

* fixed #5: fixed bug for set empty array on create
* fixed #9: fixed bug for null on fliedtype `number

**0.3.4**

* fixed #7: bug fix set a attribute to null will lead to a removal. For a string attribute a set to an empty string will also lead to a attribute removal

**0.3.3**

* bug fix set handling by passing empty arrays
* feature #6: added defaultfields option to table-config.

**0.3.2**

* added implicit set detection for update by testing for a object with one of the keys $add, $rem and $reset

**0.3.1**

* bugfix option.forward on find.()

**0.3.0**

* changed update behavior. Now there is not get before set.
* Event update, no longer with new and old value. Just the new value. This is a victim to the removal of get before set.
* added option conditionals to update. So you can define conditions to your update
* removed option removeMissing. Now you have to set a attribute to null to remove it.
* added forward option to .find() to define the direction of a range query

**Todos**

* handle throughput exceedwith a retry
* better check of the given config-data
* check for node.js 0.8.0

**Work in progress**

simple-dynamo is work in progress. Your ideas, suggestions etc. are very welcome.