
Amazon DynamoDB DataMapper For JavaScript

license **Apache-2.0**

This repository hosts several packages that collectively make up an object to document mapper for JavaScript applications using Amazon DynamoDB.

Getting started

The `@aws/dynamodb-data-mapper` package provides a simple way to persist and load an application's domain objects to and from Amazon DynamoDB. When used together with the decorators provided by the `@aws/dynamodb-data-mapper-annotations` package, you can describe the relationship between a class and its representation in DynamoDB by adding a few decorators:

```
1 import {
2   attribute,
3   hashKey,
4   rangeKey,
5   table,
6 } from '@aws/dynamodb-data-mapper-annotations';
7
8 @table('table_name')
9 class MyDomainObject {
10   @hashKey()
11   id: string;
12
13   @rangeKey({defaultProvider: () => new Date()})
14   createdAt: Date;
15
16   @attribute()
17   completed?: boolean;
18 }
```

With domain classes defined, you can interact with records in DynamoDB via an instance of `DataMapper`:

```
1 import {DataMapper} from '@aws/dynamodb-data-mapper';
2 import DynamoDB = require('aws-sdk/clients/dynamodb');
3
4 const mapper = new DataMapper({
5   client: new DynamoDB({region: 'us-west-2'}), // the SDK client used
6   // to execute operations
7   tableNamePrefix: 'dev_' // optionally, you can provide a table
8   // prefix to keep your dev and prod tables separate
9 });
```

Supported operations

Using the `mapper` object and `MyDomainObject` class defined above, you can perform the following operations:

put Creates (or overwrites) an item in the table

```
1 const toSave = Object.assign(new MyDomainObject, {id: 'foo'});
2 mapper.put(toSave).then(objectSaved => {
3     // the record has been saved
4 });
```

get Retrieves an item from DynamoDB

```
1 mapper.get(Object.assign(new MyDomainObject, {id: 'foo', createdAt: new
    Date(946684800000)}))
2     .then(myItem => {
3         // the item was found
4     })
5     .catch(err => {
6         // the item was not found
7     })
```

NB: The promise returned by the mapper will be rejected with an `ItemNotFoundException` if the item sought is not found.

update Updates an item in the table

```
1 const myItem = await mapper.get(Object.assign(
2     new MyDomainObject,
3     {id: 'foo', createdAt: new Date(946684800000)}
4 ));
5 myItem.completed = true;
6
7 await mapper.update(myItem);
```

delete Removes an item from the table

```
1 await mapper.delete(Object.assign(
2     new MyDomainObject,
3     {id: 'foo', createdAt: new Date(946684800000)}
4 ));
```

scan Lists the items in a table or index

```
1 for await (const item of mapper.scan(MyDomainObject)) {
2     // individual items will be yielded as the scan is performed
3 }
4
5 // Optionally, scan an index instead of the table:
6 for await (const item of mapper.scan(MyDomainObject, {indexName: '
    myIndex'})) {
7     // individual items will be yielded as the scan is performed
8 }
```

query Finds a specific item (or range of items) in a table or index

```
1 for await (const foo of mapper.query(MyDomainObject, {id: 'foo'})) {
2     // individual items with a hash key of "foo" will be yielded as the
    query is performed
3 }
```

Batch operations The mapper also supports batch operations. Under the hood, the batch will automatically be split into chunks that fall within DynamoDB's limits (25 for `batchPut` and `batchDelete`, 100 for `batchGet`). The items can belong to any number of tables, and exponential backoff for unprocessed items is handled automatically.

batchPut Creates (or overwrites) multiple items in the table

```
1 const toSave = [
2     Object.assign(new MyDomainObject, {id: 'foo', completed: false}),
3     Object.assign(new MyDomainObject, {id: 'bar', completed: false})
4 ];
5 for await (const persisted of mapper.batchPut(toSave)) {
6     // items will be yielded as they are successfully written
7 }
```

batchGet Fetches multiple items from the table

```
1 const toGet = [
2     Object.assign(new MyDomainObject, {id: 'foo', createdAt: new Date
    (946684800000)}),
3     Object.assign(new MyDomainObject, {id: 'bar', createdAt: new Date
    (946684800001)})
4 ];
5 for await (const found of mapper.batchGet(toGet)) {
6     // items will be yielded as they are successfully retrieved
7 }
```

```
7 }
```

NB: Only items that exist in the table will be retrieved. If a key is not found, it will be omitted from the result.

batchDelete Removes multiple items from the table

```
1 const toRemove = [  
2   Object.assign(new MyDomainObject, {id: 'foo', createdAt: new Date  
    (946684800000)}),  
3   Object.assign(new MyDomainObject, {id: 'bar', createdAt: new Date  
    (946684800001)})  
4 ];  
5 for await (const found of mapper.batchDelete(toRemove)) {  
6   // items will be yielded as they are successfully removed  
7 }
```

Operations with Expressions

Application example

```
1 import {  
2   AttributePath,  
3   FunctionExpression,  
4   UpdateExpression,  
5 } from '@aws/dynamodb-expressions';  
6  
7 const expr = new UpdateExpression();  
8  
9 // given the anotation bellow  
10 @table('tableName')  
11 class MyRecord {  
12   @hashKey()  
13   email?: string;  
14  
15   @attribute()  
16   passwordHash?: string;  
17  
18   @attribute()  
19   passwordSalt?: string;  
20  
21   @attribute()  
22   verified?: boolean;  
23  
24   @attribute()  
25   verifyToken?: string;  
26 }  
27
```

```
28 // you make a mapper operation as follows
29 const aRecord = Object.assign(new MyRecord(), {
30     email,
31     passwordHash: password,
32     passwordSalt: salt,
33     verified: false,
34     verifyToken: token,
35 });
36 mapper.put(aRecord, {
37     condition: new FunctionExpression('attribute_not_exists', new
        AttributePath('email')
38 }).then( /* result handler */ );
```

Table lifecycle operations

createTable Creates a table for the mapped class and waits for it to be initialized:

```
1 mapper.createTable(MyDomainObject, {readCapacityUnits: 5,
    writeCapacityUnits: 5})
2     .then(() => {
3         // the table has been provisioned and is ready for use!
4     })
```

ensureTableExists Like `createTable`, but only creates the table if it doesn't already exist:

```
1 mapper.ensureTableExists(MyDomainObject, {readCapacityUnits: 5,
    writeCapacityUnits: 5})
2     .then(() => {
3         // the table has been provisioned and is ready for use!
4     })
```

deleteTable Deletes the table for the mapped class and waits for it to be removed:

```
1 await mapper.deleteTable(MyDomainObject)
```

ensureTableNotExists Like `deleteTable`, but only deletes the table if it exists:

```
1 await mapper.ensureTableNotExists(MyDomainObject)
```

Constituent packages

The DataMapper is developed as a monorepo using [lerna](#). More detailed documentation about the mapper's constituent packages is available by viewing those packages directly.

- Amazon DynamoDB Automarshaller
- Amazon DynamoDB Batch Iterator
- Amazon DynamoDB DataMapper
- Amazon DynamoDB DataMapper Annotations
- Amazon DynamoDB Data Marshaller
- Amazon DynamoDB Expressions
- Amazon DynamoDB Query Iterator