

```
import "github.com/guregu/dynamo"
```

dynamo is an expressive DynamoDB client for Go, with an easy but powerful API. dynamo integrates with the official AWS SDK.

This library is stable and versioned with Go modules.

Example

```
1 package dynamo
2
3 import (
4     "time"
5
6     "github.com/aws/aws-sdk-go/aws"
7     "github.com/aws/aws-sdk-go/aws/session"
8     "github.com/guregu/dynamo"
9 )
10
11 // Use struct tags much like the standard JSON library,
12 // you can embed anonymous structs too!
13 type widget struct {
14     UserID int // Hash key, a.k.a. partition key
15     Time   time.Time // Range key, a.k.a. sort key
16
17     Msg      string `dynamo:"Message"` // Change name
18             in the database
19     Count    int `dynamo:",omitempty"` // Omits if
20             zero value
21     Children []widget // List of maps
22     Friends  []string `dynamo:",set"` // Sets
23     Set       map[string]struct{} `dynamo:",set"` // Map sets, too!
24     SecretKey string `dynamo:"-"` // Ignored
25 }
26
27 func main() {
28     sess := session.Must(session.NewSession())
29     db := dynamo.New(sess, &aws.Config{Region: aws.String("us-west-2")})
30
31     table := db.Table("Widgets")
32
33     // put item
34     w := widget{UserID: 613, Time: time.Now(), Msg: "hello"}
35     err := table.Put(w).Run()
36
37     // get the same item
```

```
36     var result widget
37     err = table.Get("UserID", w.UserID).
38         Range("Time", dynamo.Equal, w.Time).
39         One(&result)
40
41     // get all items
42     var results []widget
43     err = table.Scan().All(&results)
44
45     // use placeholders in filter expressions (see Expressions section
46         below)
47     var filtered []widget
48     err = table.Scan().Filter("'Count' > ?", 10).All(&filtered)
49 }
50 }
```

Expressions

dynamo will help you write expressions used to filter results in queries and scans, and add conditions to puts and deletes.

Attribute names may be written as is if it is not a reserved word, or be escaped with single quotes (' '). You may also use dollar signs (\$) as placeholders for attribute names and list indexes. DynamoDB has very large amount of reserved words so it may be a good idea to just escape everything.

Question marks (?) are used as placeholders for attribute values. DynamoDB doesn't have value literals, so you need to substitute everything.

Please see the DynamoDB reference on expressions for more information. The Comparison Operator and Function Reference is also handy.

```
1 // Using single quotes to escape a reserved word, and a question mark
2 // as a value placeholder.
3 // Finds all items whose date is greater than or equal to lastUpdate.
4 table.Scan().Filter("'Date' >= ?", lastUpdate).All(&results)
5
6 // Using dollar signs as a placeholder for attribute names.
7 // Deletes the item with an ID of 42 if its score is at or below the
8 // cutoff, and its name starts with G.
9 table.Delete("ID", 42).If("Score <= ? AND begins_with($, ?)", cutoff, "
10 Name", "G").Run()
11
12 // Put a new item, only if it doesn't already exist.
13 table.Put(item{ID: 42}).If("attribute_not_exists(ID)").Run()
```

Encoding support

dynamo automatically handles the following interfaces:

- `dynamo.Marshaler` and `dynamo.Unmarshaler`
- `dynamodbattribute.Marshaler` and `dynamodbattribute.Unmarshaler`
- `encoding.TextMarshaler` and `encoding.TextUnmarshaler`

This allows you to define custom encodings and provides built-in support for types such as `time.Time`.

Struct tags and fields

dynamo handles struct tags similarly to the standard library `encoding/json` package. It uses `dynamo` for the struct tag's name, taking the form of: `dynamo:"attributeName,option1,option2,etc"`. You can omit the attribute name to use the default: `dynamo:",option1,etc"`.

Renaming By default, dynamo will use the name of your fields as the name of the DynamoDB attribute it corresponds to. You can specify a different name with the `dynamo` struct tag like so: `dynamo:"other_name_goes_here"`. If two fields have the same name, dynamo will prioritize the higher-level field.

Omission If you set a field's name to `"-"` (as in `dynamo:"-"`) that field will be ignored. It will be omitted when marshaling and ignored when unmarshaling. Also, fields that start with a lowercase letter will be ignored. However, embedding a struct whose type has a lowercase letter but contains uppercase fields is OK.

Sets By default, slices will be marshaled as DynamoDB lists. To marshal a field to sets instead, use the `dynamo:",set"` option. Empty sets will be automatically omitted.

You can use maps as sets too. The following types are supported:

- `[]T`
- `map[T]struct{}`
- `map[T]bool`

where `T` represents any type that marshals into a DynamoDB string, number, or binary value.

Note that the order of objects within a set is undefined.

Omitting empty values (omitempty) Using the **omitempty** option (as in `dynamo:",omitempty"`) will omit the field if it has a zero (ex. an empty string, 0, nil pointer) value. Structs are supported.

It also supports the `isZeroer` interface below:

```
1 type isZeroer interface {  
2     IsZero() bool  
3 }
```

If `IsZero()` returns true, the field will be omitted. This gives us built-in support for `time.Time`.

You can also use the `dynamo:",omitemptyelem"` option to omit empty values inside of slices.

Automatic omission Some values will be automatically omitted.

- Empty strings
- Empty sets
- Empty structs
- Nil pointers and interfaces
- Types that implement `encoding.TextMarshaler` and whose `MarshalText` method returns 0-length or nil slice.
- Zero-length binary (byte slices)

To override this behavior, use the `dynamo:",allowempty"` flag. Not all empty types can be stored by DynamoDB. For example, empty sets will still be omitted.

To override auto-omit behavior for children of a map, for example `map[string]string`, use the `dynamo:",allowemptyelem"` option.

Using the NULL type DynamoDB has a special NULL type to represent null values. In general, this library avoids marshaling things as NULL and prefers to omit those values instead. If you want empty/nil values to marshal to NULL, use the `dynamo:",null"` option.

Unix time By default, `time.Time` will marshal to a string because it implements `encoding.TextMarshaler`.

If you want `time.Time` to marshal as a Unix time value (number of seconds since the Unix epoch), you can use the `dynamo:",unixtime"` option. This is useful for TTL fields, which must be Unix time.

Creating tables

You can use struct tags to specify hash keys, range keys, and indexes when creating a table.

For example:

```
1 type UserAction struct {
2     UserID string    `dynamo:"ID,hash" index:"Seq-ID-index,range"`
3     Time   time.Time  `dynamo:",range"`
4     Seq    int64       `localIndex:"ID-Seq-index,range" index:"Seq-ID-
        index,hash"`
5     UUID   string     `index:"UUID-index,hash"`
6 }
```

This creates a table with the primary hash key ID and range key Time. It creates two global secondary indices called UUID-index and Seq-ID-index, and a local secondary index called ID-Seq-index.

Retrying

Requests that fail with certain errors (e.g. `ThrottlingException`) are automatically retried. Methods that take a `context.Context` will retry until the context is canceled. Methods without a context will use the `RetryTimeout` global variable, which can be changed; using context is recommended instead.

Limiting or disabling retrying The maximum number of retries can be configured via the `MaxRetries` field in the `*aws.Config` passed to `dynamo.New()`. A value of 0 will disable retrying. A value of -1 means unlimited and is the default (however, context or `RetryTimeout` will still apply).

```
1 db := dynamo.New(session, &aws.Config{
2     MaxRetries: aws.Int(0), // disables automatic retrying
3 })
```

Custom retrying logic If a custom `request.Retryer` is set via the `Retryer` field in `*aws.Config`, dynamo will delegate retrying entirely to it, taking precedence over other retrying settings. This allows you to have full control over all aspects of retrying.

Example using `client.DefaultRetryer`:

```
1 retryer := client.DefaultRetryer{
2     NumMaxRetries:    10,
3     MinThrottleDelay: 500 * time.Millisecond,
4     MaxThrottleDelay: 30 * time.Second,
```

```
5 }
6 db := dynamo.New(session, &aws.Config{
7     Retryer: retryer,
8 })
```

Compatibility with the official AWS library

dynamo has been in development before the official AWS libraries were stable. We use a different encoder and decoder than the dynamodbattribute package. dynamo uses the `dynamo` struct tag instead of the `dynamodbav` struct tag, and we also prefer to automatically omit invalid values such as empty strings, whereas the dynamodbattribute package substitutes null values for them. Items that satisfy the `dynamodbattribute.(Un)marshaler` interfaces are compatible with both libraries.

In order to use dynamodbattribute's encoding facilities, you must wrap objects passed to dynamo with `dynamo.AWSEncoding`. Here is a quick example:

```
1 // Notice the use of the dynamodbav struct tag
2 type book struct {
3     ID    int    `dynamodbav:"id"`
4     Title string `dynamodbav:"title"`
5 }
6 // Putting an item
7 err := db.Table("Books").Put(dynamo.AWSEncoding(book{
8     ID:    42,
9     Title: "Principia Discordia",
10 })).Run()
11 // When getting an item you MUST pass a pointer to AWSEncoding!
12 var someBook book
13 err := db.Table("Books").Get("ID", 555).One(dynamo.AWSEncoding(&
14     someBook))
```

Integration tests

By default, tests are run in offline mode. In order to run the integration tests, some environment variables need to be set.

To run the tests against DynamoDB Local:

```
1 # Use Docker to run DynamoDB local on port 8880
2 docker compose -f '.github/docker-compose.yml' up -d
3
4 # Run the tests with a fresh table
5 # The tables will be created automatically
6 # The '%' in the table name will be replaced the current timestamp
```

```
7 DYNAMO_TEST_ENDPOINT='http://localhost:8880' \  
8   DYNAMO_TEST_REGION='local' \  
9   DYNAMO_TEST_TABLE='TestDB-%' \  
10  AWS_ACCESS_KEY_ID='dummy' \  
11  AWS_SECRET_ACCESS_KEY='dummy' \  
12  AWS_REGION='local' \  
13  go test -v -race ./... -cover -coverpkg=./...
```

License

BSD 2-Clause