HumbleJS Documentation

Release 2.2.0

Jacob Alheid

Sep 20, 2017

Contents

HumbleJS is the sister project of HumbleDB, an ODM for Python. It attempts to achieve parity as best as possible with the HumbleDB project, while leveraging nuances of the Javascript language to provide a convenient and concise API.

Contributors

- shakefu (creator, maintainer)
- nigelkibodeaux

User's Guide

CHAPTER 1

Tutorial

This is the comprehensive tutorial for using HumbleJS.

Quickstart

Install HumbleJS:

```
$ npm install --save humblejs
```

Create a new database connection:

```
var humblejs = require('humblejs');

var my_db = new humblejs.Database('mongodb://localhost:27107/my_db')
```

Your database object has a factory function for declaring document collection classes:

```
// my_db.document(collection_name, schema)
var Coder = my_db.document('coders', {
   name: 'n',
   lang: 'l',
   skill: 's'
   votes: 'v'
   })
```

The schema is used to map long class and instance property names to short document keys.

Once created, you can use your document class to create, save, find, etc. All the MongoDB collection methods are available:

```
// Create a new document instance
var doc = new Coder()
doc.name = "Grace Hopper"
doc.lang = "COBOL"

// Save the document
// Document.save(document, callback)
Coder.save(doc, function(err, doc) {
   if (err) { throw err; }
   console.log("Coder saved")
   })

// Find a document
// Queries are automatically translated from long properties to short keys
// Document.find(query, callback)
Coder.find({lang: "COBOL"}, function(err, docs) {
   if (err) { throw err; }
   docs.forEach(function(doc){ console.log(doc); })
})
```

HumbleJS also provides convenience methods for documents which define an `_id` already. If the `_id` is missing, then these will throw an error:

```
var doc = new Coder()
doc._id = 1
doc.name = "Ada Lovelace"

doc.save(function(err, doc) {
   if (err) { throw err; }
   console.log("Coder saved")
   })
```

HumbleJS also provides a way to map embedded documents:

```
var Embed = humblejs.Embed

// Embed(key, schema)
var Library = my_db.document('libraries', {
   name: '_id',
   lang: 'l',
   meta: Embed('m', {
      created: 'c',
      author: 'a'
      }),
   install: 'i'
```

```
   })

var doc = new Library()
doc.name = 'humblejs'
doc.lang = 'coffeescript'
doc.meta.created = new Date()
doc.meta.author = "Jacob Alheid"
doc.install = "npm install humblejs"

doc.insert(function (err, doc){
   if (err) { throw err; }
   console.log("Library inserted")
   })
```

See the rest of the tutorial for more features and detailed descriptions.

Installation

HumbleJS is available on npmjs.org. To install, simply run `npm install humblejs --save`.

Alternatively, you can install the latest development version directly, with:

```
$ git clone git@github.com:aboutdotme/humblejs.git
$ cd humblejs
$ npm link
```

Database connnections

This section describes database objects and their use. See the *`Database()`* API documentation for the full reference.

HumbleJS Database instances are thin wrappers around mongojs connection instances. They provide a convenience collection method as well as a factory method for Document declarations.

Example: Creating new database instances

```
var humblejs = require('humblejs');

// Create a new database with default settings (localhost:27017)
var my_db = new humblejs.Database('my_db');

// Databases can take a MongoDB connection URI
var other_db = new humblejs.Database('mongodb://db.myhost.com:30000/other');
```

Once a database is created, you can use it as an easy handle to access collections directly, or to create new *`Document()`* declarations.

Accessing a collection is done via the `Database.collection()` method. This will return a direct reference to the underlying mongojs collection instance.

Example: Accessing collections

```
var humblejs = require('humblejs');

var my_db = new humblejs.Database('my_db');

// This will return a direct reference to the underlying mongojs collection
var blog_posts = my_db.collection('blog_posts');

blog_posts.find(...) // All your collection methods are there
```

A database instance also provides a factory function for creating new document declarations. This is just a bit of syntactic sugar if you want to use it.

Example: Declaring documents in a database

```
var humblejs = require('humblejs');

var my_db = new humblejs.Database('my_db');

// This creates a new BlogPost class which stores documents in the
// ``'blog_posts'`` collection in the ``'my_db'`` database.
var BlogPost = my_db.document('blog_posts', {
   author: 'a',
   title: 't',
   body: 'b',
   published: 'p'
   });

// Otherwise it's just a normal Document class
var post = new BlogPost();
post.author = 'shakefu';
post.title = "How to use the document declaration factory";
post.body = "See the documentation.";
post.published = new Date();
post.save();
```

Documents

This section describes how to declare, instantiate, and manipulate documents.

HumbleJS documents allow you to map class and instance attributes to document keys and values, respectively. This can be very convenient since shorter document keys saves overhead on document size, but long and clear attribute names allow for very readable code.

See the *`Document()`* documentation for full reference.

Example: A basic document declaration

```
var humblejs = require('humblejs');

// Documents need a collection instance
```

```
var my_db = new humblejs.Database('my_db');

// For the sake of example, we'll get the collection directly
var blog_posts = my_db.collection('blog_posts');

// Declare a new Document subclass and its mapping
var BlogPost = new humblejs.Document(blog_posts, {
   author: 'a',
   title: 't',
   body: 'b',
   published: 'p'
   });
```

What's going on here? Well, the first argument to the *Document()* constructor is a *collection* instance. The second argument is the document schema, or attribute mapping.

Within the document schema object, its keys (`'author'`, `'title'`, etc.) will become attributes on the *BlogPost* class, and its values (`'a'`, `'t'`, etc.) will be used as the document keys when actually storing the document to the database.

Using a document schema is entirely optional - if you want to simply have the document instance attributes have the same name as the stored document keys, it can be omitted entirely.

On the document subclass itself, if an attribute is mapped (e.g. it's part of the document schema), accessing that attribute will return the key. This is for the convenience of being able to use the attribute names to reference keys in things like queries and updates. In the example above, `BlogPost.author` has the value `'a'`.

On instances of the document subclass, if an attribute is mapped, it will return the value of that key in the document or store a value to that key on assignment. So if I create a `new BlogPost()` instance, I can assign to attributes like `post.author = 'John'`, and that would translate to setting the `post['a'] = 'John'` key in the document.

Example: Working with document attributes

```
// Using the BlogPost class from the above example

// Let's create a new document instance
var post = new BlogPost();

// You can use attribute assignment for the mapped attributes
post.author = 'John Smith';

// This is the same as key assignment on the document
post['a'] = 'John Smith';

// Likewise attribute retrieval lets you access mapped keys, so
// post.author === 'John Smith'

// Only the key is stored - the attribute only exists as a convenience on
// the instance so:
// post === {a: 'John Smith'}

// When querying for documents, you can use the key directly
BlogPost.find({a: 'John Smith'}, function (err, docs) {
   // ...
   });

// Mapped class attributes return document keys, so
```

```
// BlogPost.author === 'a'
// BlogPost.title === 't'
// ... and so on

// You can use the mapped attribute in queries, making your code more
// legible, though more verbose
var query = {};
query[BlogPost.author] = 'John Smith';
BlogPost.find(query, function (err, docs){
   // ...
   });

// If `humblejs.auto_map_queries` is true, which is the default, then mapped
// attributes can be used directly in query objects, and will be
// automatically translated to their document keys
BlogPost.find({author: 'John Smith'}, function (err, docs){
   // ...
   });
```

See the section on *Document mapping* for a more in depth discussion of how mapping and auto mapping queries works.

Default values

This section describes how to provide default values.

One of the advantages of mapping attributes, even to the same key, is that HumbleJS allows you to provide default values in the case that a document is missing a key.

A default value is specified with an array in the document mapping, like `[key, default_value]` instead of just specifying a *key*.

There are two types of default values, static and dynamic. Dynamic default values are generates from the return value of a specified function. Static values are specified inline.

If you provide a static default value, that value will be returned when accessing the attribute, but not stored to the document.

If you provide a dynamic default value, when that attribute is access, the value will be stored to the document. It's up to you whether to persist the dynamic value or not.

If there are default values set in a document, they will be automatically included in the output from that document's *forJson()* method.

Example: Static and dynamic default values

```
var humblejs = require('humblejs');

var my_db = new humblejs.Database('my_db');

// We're using the document class factory here since it's convenient
var BlogPost = my_db.document('blog_posts', {
   author: 'a'
   title: 't'
   body: 'b'
```

```
   // This is a static default - until a value is specified on the document,
   // it will read as `false`, and it will not be stored in the database
   published: ['p', false]

   // This is a dynamic default - the first time `created` is accessed, the
   // function `Date.now()` will be called, and its return value will be
   // stored to the document instance
   created: ['c', Date.now]
   });

var post = new BlogPost();

// Accessing the static default doesn't change the document
post.published // === false, post === {}

// Accessing the dynamic default does change the document, only once
post.created // === Date.now(), post === {c: Date.now()}

// On subsequent accesses of an attribute with a dynamic default, the stored
// value will be returned, ensuring consistency
post.created // === <whatever time was originally returned above>

// And the dynamic value can be saved
post.save() // Accessing post.created for this document instance won't change
```

Embedded documents

This section describes how to use embedded document schemas.

HumbleJS provides a convenience object for mapping the internals of embedded documents into document properties.

Example: Basic document mapping

```
var humblejs = require('humblejs');

// Create a new database instance
var my_db = new humblejs.Database('mongodb://localhost:27017/my_db');

// Use the document factory to declare a new Document subclass
var MyDoc = my_db.document('my_docs_collection', {
    doc_id: '_id',
    value: 'val',
    meta: humblejs.Embed('meta', {
        author: 'auth',
        created: 'created'
    })
});

// The Embed class allows for sub-properties to be mapped onto document keys,
// even without assigning the parent property to an object first
var doc = new MyDoc();
doc.meta.author = "Jimmy Page";
doc.meta.created = new Date();
```

Embedded arrays

This section describes how embedded arrays work.

Document mapping

This section describes how HumbleJS maps property names to document keys.

Auto and manual mapping

This section describes how auto-mapping queries works and how to map an arbitrary long property document to short key names.

Reverse mapping

This section describes how to translate documents to a human readable or JSON friendly form.

Convenience methods

This section describes shortcut methods available on document instances.

Sparse Reports

This section describes how to use SparseReport subclasses.

CHAPTER 2

API Documentation

This section contains documentation on the public HumbleJS API.

Database

This is a helper class for managing database connections, getting collections and creating new documents.

class **`Database`** (*mongodb_uri*[, *options*])

Arguments

- **`mongodb_uri`** (*`string`*) – A MongoDB connection URI (see the MongoDB documentation on connection strings)
- **`options`** (*`object`*) – Additional connection options

`document` (*collection*[, *schema*])

Factory function for declaring new documents which belong to this database.

Arguments

- **`collection`** (*`String`*) – Collection name
- **`schema`** (*`Object`*) – Document schema

`collection` (*name*)

Return a reference to a collection *name* instance.

Arguments

- **`name`** (*`String`*) – Collection name

Document

This is the basic document class.

class **Document** (*collection*[, *schema*])

Arguments

- **collection** (*object*) – A MongoJS collection instance
- **schema** (*object*) – The schema for this document

Document subclass instances have the following methods:

forJson ([*allowDefault*])
Return a representation of this document suitable for JSON serialization. If there are default values defined for keys at the highest level of the document, they will automatically be included in the JSON representation.

If the optional *allowDefault* argument is falsey, then default values will not be included.

save ([*callback*])
Save the document to the database. If there is no *_id* field, one will created.

The optional *callback* argument may be required depending on your write concern.

insert ([*callback*])
Insert the document in the database. If there is no *_id* field, one will created.

The optional *callback* argument may be required depending on your write concern.

update (*update*[, *callback*])
Update the document with *update* clause.

If there is no *_id* field present, this will throw an error.

The optional *callback* argument may be required depending on your write concern.

remove ([*callback*])
Remove the document from the collection.

If there is no *_id* field present, this will throw an error.

The optional *callback* argument may be required depending on your write concern.

Embed

This is used to define embedded document schemas.

class **Embed** (*key*, *schema*)

Arguments

- **key** (*string*) – The key name for this embedded document
- **schema** (*object*) – The embedded document schema

SparseReport

Create a new SparseReport subclass.

A SparseReport is also a *Document()* subclass and has the same available methods.

class **SparseReport** (*collection[, schema][, options]*)

Arguments

- **collection** (*object*) – A MongoJS collection instance
- **schema** (*object*) – The document schema
- **options** (*object*) – SparseReport options
- **options.period** (*string*) – Period for this report
- **options.ttl** (*string*) – Time until a document expires
- **options.id_mark** (*string*) – Separator used in creating _id fields
- **options.sum** (*string*) – Whether to sum subkeys

record (*identifier, events[, timestamp][, callback]*)

Record an event.

Arguments

- **identifier** (*string*) – An event category or parent identifier
- **| integer events** (*object*) – Events object or an integer to increment the total by
- **timestamp** (*Date*) – Timestamp for the event (optional)
- **callback** (*function*) – Callback method

An event can have metadata attached to it, for example, when tracking signup clicks, you might record the source of the click and the plan like so:

```
var events = {
   'source.banner': 3,
   'plan.gold': 2
}
myReport.record('signup_clicks', events)
```

In the above example, the total number of 'signup_clicks' is increased by 3 (the highest number of all events recorded). Passing an empty events object increases the total 'signup_clicks' by one:

```
myReport.record('signup_clicks', {})
```

To record 13 'signup_clicks' with no metadata, you would do it like this:

```
myReport.record('signup_clicks', 13)
```

MINUTE

Period covering one minute.

HOUR

Period covering one hour.

DAY

Period covering one day.

WEEK

Period covering one week.

MONTH

Period covering one month.

YEAR

Period covering one year.

CHAPTER 3

Changelog

This section contains a brief history of changes by version.

2.x

As of version 2.0, the Changelog is maintained via GitHub Releases. Please view it there.

Pre-2.0

1.1.0

- *forJson* now takes an optional single argument, *allowDefault*, which when set to a *false*-y value will not include default values in the JSON output.

Released January 19, 2016

1.0.6

- Fix a bug with *forJson* when an *Embed* key is assigned a non-object value. Thanks to nigelkibodeaux.

Released July 29, 2015

1.0.5

- Updated to latest mongojs dependency.

1.0.1

- Fix bug with *dateRange* that was causing empty and non-empty *.all* fields to have differing lengths, as well as better test coverage. Thanks to nigelkibodeaux.

1.0.0

- Start 1.0 line.
- Change `SparseReport()` to take a *total* value and fix a bug where the total wasn't incrementing correctly when specifying values. Thanks to nigelkibodeaux. This may be backwards incompatible.

Pre-1.0

0.0.15

- *0.0.14* was a bad build.

0.0.14

- Fix a bug where dotted notation key names were not correctly mapping when there was a default value. Thanks to nigelkibodeaux for the report.

Released December 1, 2014.

0.0.13

- Fix a bug where query mapping helpers incorrectly mapped keys with default values. Thanks to nigelkibodeaux for the report.

Released November 21, 2014.

0.0.12

- Default values now work with embedded documents on document instances and when using `forJson()`.

Released November 19, 2014.

0.0.11

- When calling `forJson()`, default values defined at the top level will be included. Embedded document default values don't work yet.

0.0.10

- Fix a bug on `dateRange()` that was breaking things.

0.0.9

- New *SparseReport()* class for recording and query data aggregations.

0.0.8

- Auto map projections and update clauses.

Released September 24, 2014.

0.0.7

- Fix bug where projections were lost when calling methods synchronously.

Released September 24, 2014.

0.0.6

- Started documentation

Index