FREE eBook

LEARNING mongoose

Free unaffiliated eBook created from
Stack Overflow contributors.

#mongoose

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: mongoose

It is an unofficial and free mongoose ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official mongoose.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with mongoose

Remarks

> **Mongoose** is a **MongoDB** object modeling tool designed to work in an asynchronous environment.
>
> Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

Mongoose makes it painlessly easy to work with MongoDB database.

We can easily structure our database using `Schemas and Models`, Automate certain things when record is added or updated using `Middlewares/Hooks` and easily get the data we need by `querying` our models.

Important Links

- Mongoose Quickstart
- Mongoose GitHub Repository
- Mongoose Documentation
- Mongoose Plugins

Versions

Latest release: Version **4.6.0** released on **2nd September 2016**

All versions can be found at https://github.com/Automattic/mongoose/blob/master/History.md

Version	Release Date
1.0.1	2011-02-02
1.1.6	2011-03-22
1.3.0	2011-04-19
1.3.1	2011-04-27
1.3.4	2011-05-17
1.4.0	2011-06-10
1.5.0	2011-06-27
1.6.0	2011-07-07
2.0.0	2011-08-24

Version	Release Date
2.3.4	2011-10-18
2.5.0	2012-01-26
3.0.0	2012-08-07
3.1.2	2012-09-10
3.2.0	2012-09-27
3.5.0	2012-12-10
3.5.6	2013-02-14
3.6.0	2013-03-18
3.6.5	2013-04-15
3.8.0	2013-10-31
3.8.10	2014-05-20
3.8.15	2014-08-17
4.0.0	2015-03-25
4.0.6	2015-06-21
4.1.0	2015-07-24
4.2.0	2015-10-22
4.2.10	2015-12-08
4.3.5	2016-01-09
4.4.0	2016-02-02
4.4.4	2016-02-17
4.4.8	2016-03-18
4.4.13	2016-04-21
4.4.18	2016-05-21
4.5.0	2016-06-13
4.5.5	2016-07-18

Version	Release Date
4.5.8	2016-08-01
4.5.9	2016-08-14
4.5.10	2016-08-23
4.6.0	2016-09-02

Examples

Installation

Installing **mongoose** is as easy as running the `npm` command

```
npm install mongoose --save
```

But make sure you have also installed `MongoDB` for your OS or Have access to a MongoDB database.

Connecting to MongoDB database:

1. Import mongoose into the app:

```
import mongoose from 'mongoose';
```

2. Specify a Promise library:

```
mongoose.Promise = global.Promise;
```

3. Connect to MongoDB:

```
mongoose.connect('mongodb://127.0.0.1:27017/database');

/* Mongoose connection format looks something like this */
mongoose.connect('mongodb://USERNAME:PASSWORD@HOST::PORT/DATABASE_NAME');
```

Note:

- By default mongoose connects to MongoDB at port `27017`, Which is the default port used by MongoDB.
- To connect to MongoDB hosted somewhere else, use the second syntax. Enter MongoDB username, password, host, port and database name.

MongoDB port is 27017 by default; use your app name as the db name.

Connection with options and callback

Mongoose connect has 3 parameters, uri, options, and the callback function. To use them see sample below.

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

var uri = 'mongodb://localhost:27017/DBNAME';

var options = {
    user: 'user1',
    pass: 'pass'
}

mongoose.connect(uri, options, function(err){
    if (err) throw err;
    // if no error == connected
});
```

Read Getting started with mongoose online: https://riptutorial.com/mongoose/topic/1168/getting-started-with-mongoose

Chapter 2: Mongoose Middleware

Remarks

In mongoose, **Middlewares** are also called as `pre` and `post` hooks.

There are two types of middleware

Both of these middleware support **pre** and **post** hooks.

1. **Document middleware**

   Its supported for document functions `init`, `validate`, `save` and `remove`

2. **Query middleware**

   Its supported for query functions `count`, `find`, `findOne`, `findOneAndRemove`, `findOneAndUpdate`, `insertMany` and `update`.

Pre and Post hooks

There are two types of **Pre** hooks

1. **serial**

   As the name suggests, Its executed in serial order i..e one after another

2. **parallel**

   Parallel middleware offers more fine-grained flow control and the `hooked method` is not executed until `done` is called by all parallel middleware.

Post Middleware are executed after the `hooked method` and all of its `pre` middleware have been completed.

---

hooked methods are the functions supported by document middleware. `init, validate, save, remove`

Examples

Middleware to hash user password before saving it

This is an example of ***`Serial Document Middleware`***

In this example, We will write a middleware that will convert the plain text password into a hashed

password before saving it in database.

This middleware will automatically kick in when creating new user or updating existing user details.

FILENAME : `User.js`

```
// lets import mongoose first
import mongoose from 'mongoose'

// lets create a schema for the user model
const UserSchema = mongoose.Schema(
  {
    name: String,
    email: { type: String, lowercase: true, requird: true },
    password: String,
  },
);


/**
 * This is the middleware, It will be called before saving any record
 */
UserSchema.pre('save', function(next) {

  // check if password is present and is modified.
  if ( this.password && this.isModified('password') ) {

    // call your hashPassword method here which will return the hashed password.
    this.password = hashPassword(this.password);

  }

  // everything is done, so let's call the next callback.
  next();

});


// lets export it, so we can import it in other files.
export default mongoose.model( 'User', UserSchema );
```

Now every time we save a user, This middleware will check if password is **set** and **is modified** (this is so we dont hash users password if its not modified.)

FILENAME : `App.js`

```
import express from 'express';
import mongoose from 'mongoose';

// lets import our User Model
import User from './User';

// connect to MongoDB
mongoose.Promise = global.Promise;
mongoose.connect('mongodb://127.0.0.1:27017/database');
```

```
let app = express();
/* ... express middlewares here .... */


app.post( '/', (req, res) => {

  /*
    req.body = {
      name: 'John Doe',
      email: 'john.doe@gmail.com',
      password: '!trump'
    }
   */

  // validate the POST data

  let newUser = new User({
    name: req.body.name,
    email: req.body.email,
    password: req.body.password,
  });

  newUser.save( ( error, record ) => {
    if (error) {
      res.json({
        message: 'error',
        description: 'some error occoured while saving the user in database.'
      });
    } else {
      res.json({
        message: 'success',
        description: 'user details successfully saved.',
        user: record
      });
    }
  });

});



let server = app.listen( 3000, () => {
    console.log(`Server running at http://localhost:3000` );
  }
);
```

Read Mongoose Middleware online: https://riptutorial.com/mongoose/topic/6646/mongoose-middleware

Chapter 3: Mongoose Population

Syntax

1. Model.Query.populate(path, [select], [model], [match], [options]);

Parameters

Param	Details
path	**String** - The field key to be populated
select	**Object, String** - Field selection for the population query.
model	**Model** - Instance of the referenced model
match	**Object** - Populate conditions
options	**Object** - Query options

Examples

Simple Populate

Mongoose populate is used to show data for referenced documents from other collections.

Lets say we have a `Person` model that has referenced documents called `Address`.

Person Model

```
var Person = mongoose.model('Person', {
    fname: String,
    mname: String,
    lname: String,
    address: {type: Schema.Types.ObjectId, ref: 'Address'}
});
```

Address Model

```
var Address = mongoose.model('Address', {
    houseNum: String,
    street: String,
    city: String,
    state: String,
    country: String
});
```

To populate `Address` inside `Person` using it's ObjectId, using let's say `findOne()`, use the `populate()` function and add the field key `address` as the first parameter.

```
Person.findOne({_id: req.params.id})
    .populate('address') // <- use the populate() function
    .exec(function(err, person) {
        // do something.
        // variable `person` contains the final populated data
    });
```

Or

```
Person.findOne({_id: req.params.id}, function(err, person) {
    // do something
    // variable `person` contains the final populated data
})
.populate('address');
```

The query above should produce the document below.

Person Doc

```
{
    "_id":"123abc",
    "fname":"John",
    "mname":"Kennedy",
    "lname":"Doe",
    "address":"456def" // <- Address' Id
}
```

Address Doc

```
{
    "_id":"456def",
    "houseNum":"2",
    "street":"Street 2",
    "city":"City of the dead",
    "state":"AB",
    "country:"PH"
}
```

Populated Doc

```
 {
    "_id":"123abc",
    "fname":"John",
    "mname":"Kennedy",
    "lname":"Doe",
    "address":{
        "_id":"456def",
        "houseNum":"2",
        "street":"Street 2",
        "city":"City of the dead",
        "state":"AB",
        "country:"PH"
```

```
    }
}
```

Neglect a few fields

Let's say you **don't want** the fields `houseNum` and `street` in the `address` field of the final populated doc, use the `populate()` as follows,

```
Person.findOne({_id: req.params.id})
    .populate('address', '-houseNum -street')  // note the `-` symbol
    .exec(function(err, person) {
        // do something.
        // variable `person` contains the final populated data
    });
```

Or

```
Person.findOne({_id: req.params.id}, function(err, person) {
    // do something
    // variable `person` contains the final populated data
})
.populate('address', '-houseNum -street');  // note the `-` symbol
```

This will produce the following final populated doc,

Populated Doc

```
 {
    "_id":"123abc",
    "fname":"John",
    "mname":"Kennedy",
    "lname":"Doe",
    "address":{
        "_id":"456def",
        "city":"City of the dead",
        "state":"AB",
        "country:"PH"
    }
}
```

Populate only a few fields

If you **only want** the fields `houseNum` and `street` in the `address` field in the final populated doc, use the `populate()` function as follows in the above two methods,

```
Person.findOne({_id: req.params.id})
    .populate('address', 'houseNum street')
    .exec(function(err, person) {
        // do something.
        // variable `person` contains the final populated data
    });
```

Or

```
Person.findOne({_id: req.params.id}, function(err, person) {
    // do something
    // variable `person` contains the final populated data
})
.populate('address', 'houseNum street');
```

This will produce the following final populated doc,

Populated Doc

```
 {
    "_id":"123abc",
    "fname":"John",
    "mname":"Kennedy",
    "lname":"Doe",
    "address":{
        "_id":"456def",
        "houseNum":"2",
        "street":"Street 2"
    }
}
```

Nested Population

Lets say you have a `user` schema, which contains `name` , `contactNo`, `address`, and `friends`.

```
var UserSchema = new mongoose.Schema({
    name : String,
    contactNo : Number,
    address : String,
    friends :[{
        type: mongoose.Schema.Types.ObjectId,
        ref : User
    }]
});
```

If you want to find a user, his **friends** and **friends of friends**, you need to do population on 2 levels i.e. **nested Population**.

To find friends and friends of friends:

```
User.find({_id : userID})
    .populate({
        path : 'friends',
        populate : { path : 'friends'}//to find friends of friends
    });
```

All the `parameters` and `options` of `populate` can be used inside nested populate too, to get the desired result.

Similarly, you can `populate` more `levels` according to your requirement.

It is not recommended to do nested population for more than 3 levels. In case you need to do

nested populate for more than 3 levels, you might need to restructure your schema.

Read Mongoose Population online: https://riptutorial.com/mongoose/topic/2616/mongoose-population

Chapter 4: Mongoose Population

Syntax

- Query.populate(path, [select], [model], [match], [options])

Parameters

Parameter	Explanation
path	<**Object, String**> either the path to populate or an object specifying all parameters
[select]	<**Object, String**> Field selection for the population query (can use `'-id'` to include everything but the `id` field)
[model]	<**Model**> The model you wish to use for population.If not specified, populate will look up the model by the name in the Schema's ref field.
[match]	<**Object**> Conditions for the population
[options]	<**Object**> Options for the population query (sort, etc)

Examples

A simple mongoose populate example

`.populate()` in Mongoose allows you to populate a reference you have in your current collection or document with the information from that collection. The previous may sound confusing but I think an example will help clear up any confusion.

The following code creates two collections, User and Post:

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema

var userSchema = Schema({
  name: String,
  age: Number,
  posts: [{ type: Schema.Types.ObjectId, ref: 'Post' }]
});

var PostSchema = Schema({
  user: { type: Schema.Types.ObjectId, ref: 'User' },
  title: String,
  content: String
});
```

```
var User = mongoose.model('User', userSchema);
var Post = mongoose.model('Post', postSchema);
```

If we wanted to populate all of the posts for each user when we `.find({})` all of the Users, we could do the following:

```
User
  .find({})
  .populate('posts')
  .exec(function(err, users) {
    if(err) console.log(err);
    //this will log all of the users with each of their posts
    else console.log(users);
  })
```

Read Mongoose Population online: https://riptutorial.com/mongoose/topic/6578/mongoose-population

Chapter 5: mongoose pre and post middleware (hooks)

Examples

Middleware

Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins. Mongoose 4.0 has 2 types of middleware: document middleware and query middleware. Document middleware is supported for the following document functions.

- init
- validate
- save
- remove

Query middleware is supported for the following Model and Query functions.

- count
- find
- findOne
- findOneAndRemove
- findOneAndUpdate
- update

Both document middleware and query middleware support pre and post hooks.

Pre

There are two types of pre hooks, serial and parallel.

Serial

Serial middleware are executed one after another, when each middleware calls next.

```
var schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

Parallel

Parallel middleware offer more fine-grained flow control.

var schema = new Schema(..);

```
// `true` means this is a parallel middleware. You **must** specify `true`
// as the second parameter if you want to use parallel middleware.
schema.pre('save', true, function(next, done) {
  // calling next kicks off the next middleware in parallel
  next();
  setTimeout(done, 100);
});
```

The hooked method, in this case save, will not be executed until done is called by each middleware.

Post middleware

post middleware are executed after the hooked method and all of its pre middleware have completed. post middleware do not directly receive flow control, e.g. no next or done callbacks are passed to it. post hooks are a way to register traditional event listeners for these methods.

```
schema.post('init', function(doc) {
  console.log('%s has been initialized from the db', doc._id);
});
schema.post('validate', function(doc) {
  console.log('%s has been validated (but not saved yet)', doc._id);
});
schema.post('save', function(doc) {
  console.log('%s has been saved', doc._id);
});
schema.post('remove', function(doc) {
  console.log('%s has been removed', doc._id);
});
```

Read mongoose pre and post middleware (hooks) online:
https://riptutorial.com/mongoose/topic/4131/mongoose-pre-and-post-middleware--hooks-

Chapter 6: Mongoose Queries

Introduction

Mongoose is a Node.JS driver for MongoDB. It provides certain benefits over the default MongoDB driver, such as adding types to Schemas. One difference is that some Mongoose queries may differ from their MongoDB equivalents.

Examples

Find One Query

Import a Mongoose Model (See topic "Mongoose Schemas")

```
var User = require("../models/user-schema.js")
```

The findOne method will return the first entry in the database that matches the first parameter. The parameter should be an object where the key is the field to look for and the value is the value to be matched. This can use MongoDB query syntax, such as the dot (.) operator to search subfields.

```
User.findOne({"name": "Fernando"}, function(err, result){
    if(err) throw err;    //There was an error with the database.
    if(!result) console.log("No one is named Fernando."); //The query found no results.
    else {
        console.log(result.name); //Prints "Fernando"
    }
}
```

Read Mongoose Queries online: https://riptutorial.com/mongoose/topic/9349/mongoose-queries

Chapter 7: Mongoose Schemas

Examples

Basic Schema

A basic User Schema:

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

var userSchema = new mongoose.Schema({
    name: String,
    password: String,
    age: Number,
    created: {type: Date, default: Date.now}
});

var User = mongoose.model('User', userSchema);
```

Schema Types.

Schema methods

Methods can be set on Schemas to help doing things related to that schema(s), and keeping them well organized.

```
userSchema.methods.normalize = function() {
    this.name = this.name.toLowerCase();
};
```

Example usage:

```
erik = new User({
    'name': 'Erik',
    'password': 'pass'
});
erik.normalize();
erik.save();
```

Schema Statics

Schema Statics are methods that can be invoked directly by a Model (unlike Schema Methods, which need to be invoked by an instance of a Mongoose document). You assign a Static to a schema by adding the function to the schema's `statics` object.

One example use case is for constructing custom queries:

```
userSchema.statics.findByName = function(name, callback) {
```

```
    return this.model.find({ name: name }, callback);
}

var User = mongoose.model('User', userSchema)

User.findByName('Kobe', function(err, documents) {
    console.log(documents)
})
```

GeoObjects Schema

A generic schema useful to work with geo-objects like points, linestrings and polygons. Both **Mongoose** and **MongoDB** support **Geojson**.

Example of usage in **Node/Express**:

```
var mongoose = require('mongoose');
var Schema   = mongoose.Schema;

// Creates a GeoObject Schema.
var myGeo= new Schema({
                        name: { type: String },
                        geo : {
                                type : {
                                         type: String,
                                         enum: ['Point', 'LineString', 'Polygon']
                                },
                                coordinates : Array
                         }
});

//2dsphere index on geo field to work with geoSpatial queries
myGeo.index({geo : '2dsphere'});
module.exports = mongoose.model('myGeo', myGeo);
```

Saving Current Time and Update Time

This kind of schema will be useful if you want to keep trace of your items by insertion time or update time.

```
var mongoose = require('mongoose');
var Schema   = mongoose.Schema;

// Creates a User Schema.
var user = new Schema({
                       name: { type: String },
                       age : { type: Integer},
                       sex : { type: String },
                       created_at: {type: Date, default: Date.now},
                       updated_at: {type: Date, default: Date.now}
});

// Sets the created_at parameter equal to the current time
user.pre('save', function(next){
    now = new Date();
    this.updated_at = now;
```

```
    if(!this.created_at) {
        this.created_at = now
    }
    next();
});

module.exports = mongoose.model('user', user);
```

Read Mongoose Schemas online: https://riptutorial.com/mongoose/topic/2592/mongoose-schemas

Chapter 8: Mongoose Schemas

Examples

Creating a Schema

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

//assume Player and Board schemas are already made
var Player = mongoose.model('Player');
var Board = mongoose.model('Board');

//Each key in the schema is associated with schema type (ie. String, Number, Date, etc)
var gameSchema = new mongoose.Schema({
  name: String,
  players: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Player'
  }],
  host: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Player'
  },
  board: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Board'
  },
  active: {
    type: Boolean,
    default: true
  },
  state: {
    type: String,
    enum: ['decision', 'run', 'waiting'],
    default: 'waiting'
  },
  numFlags: {
    type: Number,
    enum: [1,2,3,4]
  },
  isWon: {
    type: Boolean,
    default: false
  }
});

mongoose.model('Game', gameSchema);
```

Read Mongoose Schemas online: https://riptutorial.com/mongoose/topic/6622/mongoose-schemas

Credits

S. No	Chapters	Contributors
1	Getting started with mongoose	4444, CENT1PEDE, Community, Delapouite, dublicator, jisoo, Random User, zurfyx
2	Mongoose Middleware	Delapouite, Random User
3	Mongoose Population	CENT1PEDE, Chinni, Medet Tleukabiluly, Ravi Shankar
4	mongoose pre and post middleware (hooks)	Naeem Shaikh
5	Mongoose Queries	Gibryon Bhojraj
6	Mongoose Schemas	AndreaM16, Ian, zurfyx