iTechArt

Software
Engineering
Services

MONGOOSE

Anton Chernov

WHAT IS MONGOOSE?

→ MongoDB object modelling for Node.js

AGENDA

1. Main inconveniences of using the native MongoDB driver for Node.js
2. Mongoose: what is it and what are its killer features:
    - ❑ Validation
    - ❑ Casting
    - ❑ Encapsulation
    - ❑ Middlewares (lifecycle management)
    - ❑ Population
    - ❑ Query building
3. Schema-Driven Design for your applications.
4. Useful links and Q&A

MAIN INCONVENIENCES OF USING THE NATIVE MONGODB DRIVER

1. No data validation
2. No casting during inserts
3. No encapsulation
4. No references(joins)

WHAT IS MONGOOSE

1. Object Data Modelling (ODM) for Node.js
2. Officially supported by MongoDB
3. Features:
   - ❑ Async and sync validation of models
   - ❑ Model casting
   - ❑ Object lifecycle management(middlewares)
   - ❑ Pseudo-joins!
   - ❑ Query builder

mongoose

```
npm install mongoose
```

MONGOOSE SETUP

CODE SNIPPET

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/mycollection');

var Schema = mongoose.Schema;

var PostSchema = new Schema({
  title: {type: String, required: true},
  body: {type: String, required: true},
  author: {type: ObjectId, required: true, ref: 'User'},
  tags: [String],
  date: {type: Date, default: Date.now}
});

mongoose.model('Post', PostSchema);
```

Validation of presence

Reference

Simplified declaration

Default value

MONGOOSE FEATURES: VALIDATION

CODE SNIPPET

Simplest validation example: presence

```
var UserSchema = new Schema({ name: { type: String, required: true } });
var UserSchema = new Schema({ name: { type: String, required: 'Oh snap! Name is 
required.' } });
```

Passing validation function:

```
var UserSchema = new Schema({ name: { type: String, validator: validate } });
var validate = function (value) { /*...*/ }; // synchronous validator
var validateAsync = function (value, respond) { respond(false); }; // async 
validator
```

Via .path() API call:

```
UserSchema.path('email').validate(function (email, respond) {
  var User = mongoose.model('User');
  User.find({email: email}).exec(function (err, users) {
    return respond(!err && users.length === 0);
  });
}, 'Such email is already registered');
```

MONGOOSE FEATURES: CASTING

1. Each property is cast to its mapped SchemaType in the document, obtained by the query.
2. Valid SchemaTypes are:
   - ❑ String
   - ❑ Number
   - ❑ Date
   - ❑ Buffer
   - ❑ Boolean
   - ❑ Mixed
   - ❑ ObjectId
   - ❑ Array

CODE SNIPPET

```
var PostSchema = new Schema({
  _id: ObjectId, // implicitly exists
  title: {type: String, required:
true},
  body: {type: String, required:
true},
  author: {type: ObjectId, required:
true, ref: 'User'},
  tags: [String],
  date: {type: Date, default:
Date.now},
  is_featured: {type: Boolean,
default: false}
});
```

MONGOOSE FEATURES: ENCAPSULATION

CODE SNIPPET

```
PostSchema.statics = {
  dropAllPosts: function (areYouSure) {
    // drop the posts!
  }
};


PostSchema.methods = {
  addComment: function (user, comment, callback) {
    // add comment here
  },
  removeComment: function (user, comment, callback) {
    // remove comment here
  }
};
```

MONGOOSE FEATURES: LIFECYCLE MANAGEMENT

CODE SNIPPET

Models have .pre() and .post() middlewares, which can hook custom functions to init, save, validate and remove model methods:

```
schema.pre('save', true, function (next, done) {
// calling next kicks off the next middleware in parallel
  next();
  doAsync(done);
});

schema.post('save', function (doc) {
  console.log('%s has been saved', doc._id);
});
```

MONGOOSE FEATURES: POPULATION

CODE SNIPPET

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s):

```
PostSchema.statics = {
  load: function (permalink, callback) {
    this.findOne({ permalink: permalink })
      .populate('author', 'username avatar_url')
      .populate('comments', 'author body date')
      .exec(callback);
  }
};
```

MONGOOSE FEATURES: QUERY BUILDING

CODE SNIPPET

Different methods could be stacked one upon the other, but the query itself will be generated and executed only after .exec():

```
var options = {
  perPage: 10,
  page: 1
};

this.find(criteria)
  .populate('author', 'username')
  .sort({'date_published': -1})
  .limit(options.perPage)
  .skip(options.perPage * options.page)
  .exec(callback);
```

SCHEMA-DRIVEN DESIGN FOR YOUR APPLICATIONS

- Schemas should match data-access patterns of your application.
- You should pre-join data where it’s possible (and use Mongoose’s .populate() wisely!).
- You have no constraints and transactions — keep that in mind.
- Using Mongoose for designing your application’s Schemas is similar to OOP design of your code.

USEFUL LINKS

- http://www.passportjs.org/Mongoose GitHub repo: https://github.com/learnboost/mongoose
- Mongoose API docs and tutorials: http://mongoosejs.com/
- MongoDB native Node.js driver docs: http://mongodb.github.io/node-mongodb-native/

Any questions?