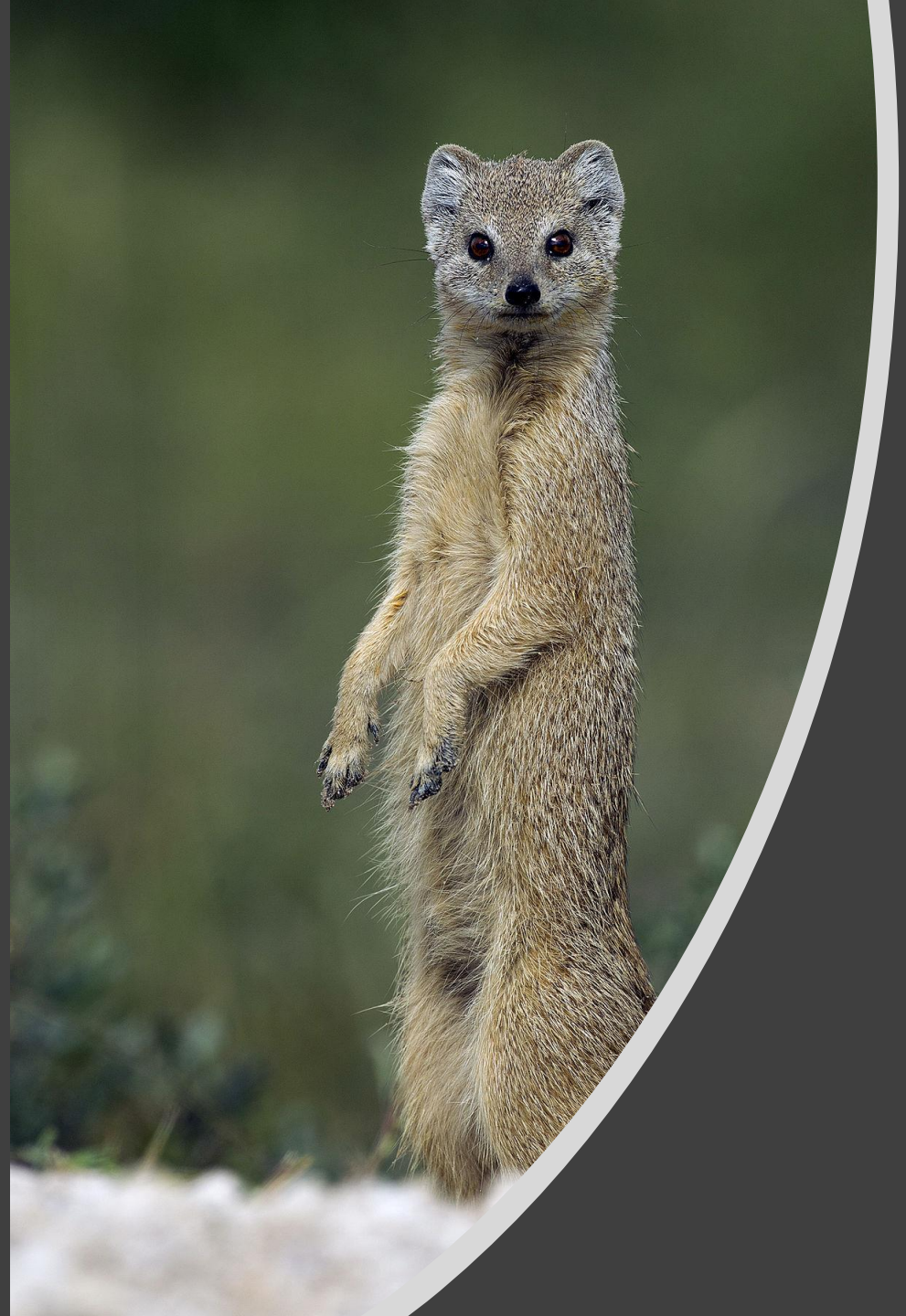

MONGOOSE

Mongo with Node.js

Mongoose Overview

- Mongoose is a object-document model module in Node.js for MongoDB
  - Wraps the functionality of the native MongoDB driver
  - Exposes models to control the records in a doc
  - Supports validation on save
  - Extends the native queries

mongoose

elegant mongodb object modeling for node.js

read the docs | discover plugins

Star 18,205 Version 5.4.19 Fork 2,570

Let's face it, **writing MongoDB validation, casting and business logic boilerplate is a drag**. That's why we wrote Mongoose.

Mongoose first?

- Shortcut to understanding the basics
- Similar to Object Relational Mapping libraries like JPA/Hibernate
- Easier concept if coming from relational DB background.

Installing & Using Mongoose

1. Run the following from the CMD/Terminal

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

2. Import the module

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

3. Connect to the database

```
mongoose.connect(process.env.mongoDB);
```

Mongoose Schemas and Models

- Mongoose supports models
  - i.e. fixed types of documents
  - Needs a mongoose.Schema
  - Each of the properties must have a type
    - Number, String, Boolean, array, object

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const UserSchema = new Schema({
  username: { type: String, unique: true, required: true},
  password: {type: String, required: true }
});

export default mongoose.model('User', UserSchema);
```

Mongoose Schemas – Arrays & sub-documents

Comments property is an Array of CommentSchemas

```
1   const mongoose = require('mongoose
2   Schema = mongoose.Schema;
3
4   const CommentSchema = new Schema({
5     body: {type: String, required:true},
6     author: {type: String, required:true},
7     upvotes:Number
8     });
9
10   const PostSchema = new Schema({
11      title: {type: String, required:true},
12      link:  {type: String, optional:true},
13      username:  {type: String, required:true},
14      comments: [CommentSchema],
15    upvotes: { type: Number, min: 0, max: 100 }
16   });
17
18  export default mongoose.model('posts', PostSchema);
```

Mongoose Schemas - Arrays

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const MovieReviewSchema = {
  userName : { type: String},
  review : {type: String}
}

const MovieSchema = new Schema({
    adult: { type: Boolean},
    id: { type: Number, required: true, unique: true },
    poster_path: { type: String},
    overview: { type: String},
    release_date: { type: String},
    reviews : [ MovieReviewSchema],
    original_title: { type: String},
    genre_ids: [{type: Number}],
```

Review property is an Array of MovieReviewSchema

Mongoose Schema – Built-in Validation

constraints on properties :

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const ContactSchema = new Schema({
  name: {type: String, required:[true, 'Name is a required property']},
  address: String,
  age: {
    type: Number,
    min: 0,
    max: 120,required: true
  },
  email: String,
  updated: {
    type: Date,
    default: Date.now,
  },
});

export default mongoose.model('Contact', ContactSchema);
```

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const UserSchema = new Schema({
  username: { type: String, unique: true, required: true},
  password: {type: String, required: true }
});

export default mongoose.model('User', UserSchema);
```

Mongoose Custom Validation

- Developers can define custom validation on their properties (e.g. validate email field is correct format)

```
ContactSchema.path('email').validate((email) => {
  var emailRegex = /^([\w-\.]+@([\w-]+\.)+[\w-]{2,4})?$/;
  return emailRegex.test(email);
}, 'A valid e-mail address is required');
```

Using Regular Expression (regex) to test for a valid email. If you've not come across them before check out https://www.w3schools.com/jsref/jsref_obj_regexp.asp

Mongoose Custom Validation

- Developers can define custom validation on their properties (e.g. validate length of username when trying to save)

```
const UserSchema = new Schema({
  username: { type: String, unique: true, required: true},
  password: {type: String, required: true },
  favourites: [{type: mongoose.Schema.Types.ObjectId,
                ref: 'Movie'  }]
});

UserSchema.path('username').validate(v => (v.length<5)?false:true)
```

Data Manipulation Mongoose

- Mongoose supports all the CRUD operations:
  - Create –> Model.create()
  - Read –> Model.find()
  - Update –> Model.update(condition, props, cb)
  - Remove –> Model.remove()
- Can operate with "*error first*" callbacks or promises.

Create with Mongoose

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const MovieReviewSchema = {
  userName : { type: String},
  review : {type: String}
}

const MovieSchema = new Schema({
    adult: { type: Boolean},
    id: { type: Number, required: true, unique: true },
    poster_path: { type: String},
    overview: { type: String},
    release_date: { type: String},
    reviews : [ MovieReviewSchema],
    original_title: { type: String},
    genre_ids: [{type: Number}],

export default mongoose.model('Movie', MovieSchema);
```

```
api
  genres
  movies
    index.js
    movieModel.js
```

```
import Movie from './movieModel';
const router = express.Router();

router.post('/', (req, res) => {
  Movie.create(req.body).then(movie => res.status(201).json(movie))
});
```

Update with Mongoose

```
// Update a user
router.put('/:id',  (req, res) => {
    if (req.body._id) delete req.body._id;
     User.update({
      _id: req.params.id,
    }, req.body, {
      upsert: false,
    })
    .then(user => res.json(200, user));
});
```

Mongoose Queries

- Mongoose provides a more expressive version of the native MongoDB
  - Instead of:
    {$or: [{conditionOne: true}, {conditionTwo: true}]
  - Do: .where({conditionOne:true}).or({conditionTwo: true})

Mongoose Queries

- Mongoose supports many queries:
  - For equality/non-equality
  - Selection of some properties
  - Sorting
  - Limit & skip
- All queries are executed over the object returned by Model.find*()
  - Model.findOne() returns a single document, the first match
  - Model.find() returns all
  - Model.findById() queries on the _id field.

```
router.post('/:userName/favourites', (req, res, next) => {
  const newFavourite = req.body;
  const userName = req.params.userName;
  if (newFavourite && newFavourite.id) {
      Movie.findOneAndUpdate({id: newFavourite.id},newFavourite,{new:true,upsert:true}).then(movie => {
          User.findByUserName(userName).then(
                  (user) => {
                     (user.favourites.indexOf(movie._id)>-1)?user:user.favourites.push(movie._id.toString());
                     user.save().then(user => res.status(201).send(user))
                    }
          );
          }).catch((err) => console.log(err));
  } else {
      res.status(401).send("unable")
  }
```

Mongoose Queries

- Can build complex queries and execute them later

```
1  const query = ContactModel.where('age').gt(17).lt(66)
2                            .where('county').in(['Waterford','Wexford','Kilkenny']);
3
4  query.exec((err,contacts)=>{...})
5
6
```

- The above finds all contacts where age >17 and <66 and living in either Waterford, Kilkenny or Wexford

Mongoose Sub-Docs

- Ex: Movies – Adding a review to a favourite movie.

```
router.post('/:id/reviews', (req, res) => {
  const id = parseInt(req.params.id);
  Movie.findByMovieDBId(id).then(movie => {
    movie.reviews.push(req.body)
    movie.save().then(res.status(200).send(movie.reviews))});
});
```