Banco de dados + Backend

Reprograma - Backend

Profa: Kelly Joany de Oliveira Santos

Object Document Mapper (ODM)



elegant mongodb object modeling for node.js

Setup

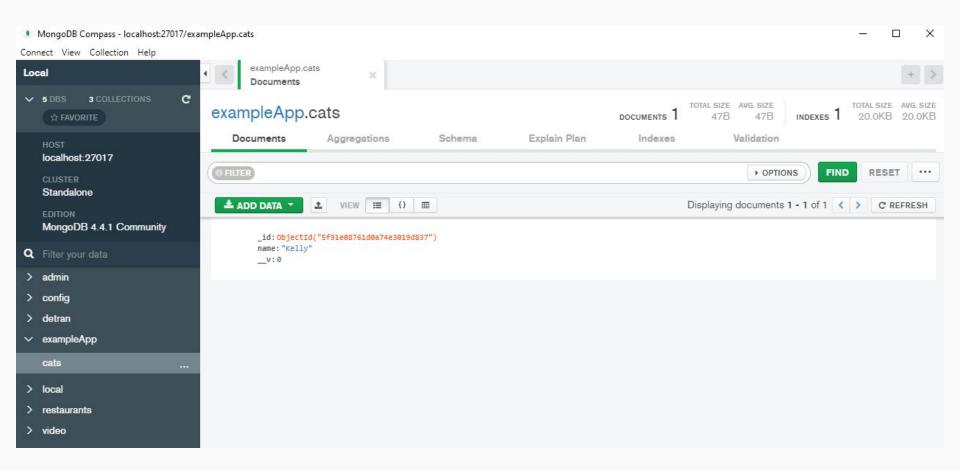
- \$ mkdir mongoose-example
- \$ cd mongoose-example
- \$ npm init
- \$ npm install mongoose
- \$ touch example.js

Connecting to the Database

```
Criar arquivo "example.js"
const mongoose = require('mongoose');
mongoose
 .connect('mongodb://localhost/exampleApp', {useNewUrlParser: true})
 .then(x => console.log(`Connected to Mongo! Database name: \$\{x.connections[0].name\}^{"`}))
 .catch(err => console.error('Error connecting to mongo', err));
```

Criar uma instância e salvar

```
const Cat = mongoose.model('Cat', { name: String });
const kitty = new Cat({ name: 'kelly' });
kitty
  .save()
  .then(newCat => console.log(`A new cat is created: ${newCat}!`))
  .catch(err => console.log(`Error while creating a new cat: ${err}`));
```



Entendendo uma callback

```
Cat.find({}, (err, cats) => {
  if(err){
    console.log(`Error occurred during getting cats from DB: ${err}`);
    return;
}
console.log('All the CATS!');
// cats is an array of Cat instances
cats.forEach(cat => console.log(` --> cat: ${cat.name}`));
});
```

O mesmo código usando Promisses

```
Cat
    .find()
    .then(catsFromDB => {
        // catsFromDB is an array of Cat instances
        catsFromDB.forEach(oneCat => console.log(` --> cat: ${oneCat.name}`));
})
    .catch(err => console.log(`Error occurred during getting cats from DB: ${err}`);)
```

Nos bastidores, o find () do Mongoose está enviando um comando find do MongoDB para o banco de dados. Como chamamos o método find () usando o modelo Cat, o Mongoose está encontrando os documentos resultantes de dentro da coleção cats.

Mongoose connection events

Connection event	Description
<pre>mongoose.connection.on('connected', callback)</pre>	Call callback when Mongoose is connected.
<pre>mongoose.connection.on('error', callback)</pre>	Call callback when an error happened on connection.
<pre>mongoose.connection.on('disconnected', callback)</pre>	Call callback when Mongoose is disconnected.
<pre>process.on('SIGINT', callback)</pre>	Call callback just before stopping Node (can be simulated with <ctrl>-C)</ctrl>

Resumindo, um Schema fornece nossa estrutura de banco de dados. Isso nos ajuda a garantir a consistência em nosso banco de dados, e temos menos probabilidade de:

- Adicionar campos que n\u00e3o deveriam existir (porque voc\u00e2 os definiu em seu esquema).
- Esquecer os campos que são obrigatórios.
- Usar o tipo errado em um campo (ou seja, um número dentro de um campo de data ou um booleano em um campo de número).

```
mongoose.model('Cat', {
  name: String
})
```

Vamos dar uma olhada em como definir um esquema adicionando alguns campos ao nosso modelo "Cat". Vamos criar a pasta "models" e adicionar um arquivo Cat.js dentro da pasta models. Vamos adicionar este snippet de código a ele:

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

const catSchema = new Schema({
   name : String,
   color: String,
   age : Number
});

const Cat = mongoose.model('Cat', catSchema);
module.exports = Cat;
```

Nosso schema é apenas um objeto com chaves para cada um dos nomes dos campos e valores para cada tipo de campo. Existem muitos tipos de schema disponíveis no Mongoose, mas aqui está uma lista rápida de tipos comuns:

- String (ex: kelly)
- Number (ex: 42)
- Date (ex: Date('2020-12-25'))
- Boolean (ex: true)
- Schema. Types. ObjectId: para armazenar ids de outras coleções
- Schema. Types. Mixed: to store anything
- Array or []: para armazenar um array de qualquer tipo (ex: ['foo', 42])
 - [String]: para armazenar um array de strings (ex: ['foo', 'bar'])
 - [Number]: para armazenar um array de numbers (ex: [42, -6])

Freqüentemente, queremos um valor padrão para algum campo, caso contrário, ele será salvo vazio. Por exemplo, quando um usuário se registra em um site e não fornece uma imagem, você pode definir a imagem como padrão para que o usuário obtenha a imagem de avatar padrão. Você pode fazer isso usando a palavra-chave "default".

```
const userSchema = new Schema({
  email : { type: String},
  username : { type: String},
  avatarUrl: { type: String, default: 'images/default-avatar.png' }
});
```

Além de especificar o tipo dos campos ou seu valor padrão, podemos definir uma validação mais detalhada definindo mais propriedades:

Field property	Possible values	Description
type	String, Number,	Sets a type for the field
default	Anything	Sets a default value
required	true Adds a required validator	
unique	true	Declares an unique index
enum	An array	Adds an enum validator
min	A number	Sets a minimum number validator
max	A number	Sets a maximum number validator
minlength	A number Sets a minimum length validator	
maxlength	A number	Sets a maximum length validator
trim	rim Adds a trim setter	
lowercase	true Adds an lowercase setter	
match	A regex	Sets a regex validator
validate	An object	Adds a custom validator (see next part)
set	A function	Adds a custom setter (see next part)

```
const catSchema = new Schema({
 name: { type: String, required: true },
 age: { type: Number, min: 0, max: 30 },
 color: { type: String, enum: ['white', 'black', 'brown'] },
  avatarUrl: { type: String, default: 'images/default-avatar.png' },
 location: {
   address: String,
   city: String
 countryCode: { type: String, match: /^[A-Z]{2}$/ },
 created: {
   type: Date,
   default: Date.now
});
```

Se você não tiver o validador certo, o Mongoose lhe dá a oportunidade de escrever o seu próprio. Para isso, você precisará adicionar "validate" e uma função "validator" e uma mensagem de erro.

```
// Example of custom validation
const userSchema = new Schema({
 linkedinProfile: {
   type: String,
    validate: {
     validator: (text) => {
        return text.indexOf('https://www.linkedin.com/') === 0;
     },
     message: "linkedinProfile must start with 'https://www.linkedin.com/'"
};
```

O Mongoose também lhe dá a oportunidade de escrever seu "setter". Cada vez que você definir um campo, o Mongoose transformará o valor através da função especificada por "set".

```
// Example of custom setter
function capitalize (val) {
 if (typeof val !== 'string') {
     val = '';
 return val.charAt(0).toUpperCase() + val.substring(1);
const userSchema = new Schema({
 name: {
    type: String,
    set: capitalize // <= here we call the setter we defined earlier
});
```

Primeiro, vamos ver como podemos inserir (create) alguns documentos. Se você der uma olhada na documentação do Mongoose, verá que pode usar a função Model.create (). Vamos ver como fica se quisermos criar um novo usuário com base no modelo de usuário já definido. Para esta demonstração, criaremos User.js dentro da pasta "models":

```
// models/User.js
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const userSchema = new Schema({
 name: { type: String },
 password: { type: String },
 job: { type: String }
 timestamps: true
});
const User = mongoose.model('User', userSchema);
module.exports = User;
```

```
User.create({ name: 'Alice', password:"ironhack2018", job: 'Architect' })
   .then(user => { console.log('The user is saved and its value is: ', user) })
   .catch(err => { console.log('An error happened:', err) });
```

Para recuperar vários documentos de um banco de dados, podemos usar Model.find () e estes são alguns dos detalhes de sua sintaxe:

- O primeiro parâmetro é a consulta representada por um objeto. Se você não colocar, encontrará todos os documentos do seu acervo;
- O segundo parâmetro é opcional e é a projeção representada por um objeto ou string com todos os campos a serem exibidos;
- O terceiro parâmetro é opcional e são as opções, representadas por um objeto;
- O último parâmetro é a função de retorno de chamada executada quando a consulta é concluída. Se você não colocar nenhuma função como parâmetro, ele retornará uma promessa.

```
let callback = (err, users) => {});
// Find all users and execute the callback
User.find({}, callback);
// Find all users where (the name is 'Alice' and the age is >= 18)
// and execute the callback
User.find({ name: 'Alice', age: { $gte: 18 }}, callback);
// Find all users where the name is 'Alice' and only selecting
//the "name" and "age" fields, and afterwards, execute the callback
User.find({ name: 'Alice' }, 'name age', callback);
// Find all users and sort them by name descending
User.find({}, null, {sort: { name: -1 }}, callback);
// Find all users where the name contains 'alice' (insensitive case)
// and execute the callback
User.find({ name: /alice/i}, callback);
// Promise version
User.find({ name: 'Alice' })
  .then(successCallback)
  .catch(errorCallback);
```

No caso de querermos recuperar apenas um documento específico, podemos usar "findOne" ou "findById"

```
User.findOne({ name: "Alice" })
  .then(successCallback)
  .catch(errorCallback);

User.findById("someMongoIdGoesHere129")
  .then(successCallback)
  .catch(errorCallback);
```

Para atualizar um campo, você pode usar "updateMany", "updateOne" ou "findByIdAndUpdate". Em cada caso, você precisa definir primeiro a condição (uma consulta ou um id) e depois a atualização, como no próximo exemplo. Decidimos aqui usar apenas a sintaxe Promise.

```
// For all users that as "@gmail\.com" in its email, change the company to "reprograma"
User.updateMany({ email: /@gmail\.com/}, { company: "reprograma" })
  .then(successCallback)
  .catch(errorCallback);
// For the first "Alice" found, change the company to "reprograma"
User.updateOne({ name: "Alice"}, { company: "reprograma" })
  .then(successCallback)
  .catch(errorCallback);
// For the user with id "5a3a7ecbc6ca8b9ce68bd41b", increment the salary by 4200
User.findByIdAndUpdate("5a3a7ecbc6ca8b9ce68bd41b", { $inc: {salary: 4200} })
  .then(successCallback)
  .catch(errorCallback);
```

Para atualizar um campo, você pode usar "deleteMany", "deleteOne" ou "findByIdAndRemove", como no próximo exemplo.

```
// Delete all the users that have "@gmail.com" in their email
User.deleteMany({ email: /@gmail\.com/})
  .then(successCallback)
  .catch(errorCallback);
// Delete the first "Alice" found
User.deleteOne({ name: "Alice"})
  .then(successCallback)
  .catch(errorCallback);
// Delete the user with id "5a3a7ecbc6ca8b9ce68bd41b"
User.findByIdAndRemove("5a3a7ecbc6ca8b9ce68bd41b")
  .then(successCallback)
  .catch(errorCallback);
```

O Mongoose também fornece alguns métodos utilitários. Provavelmente, o que você mais usará é o método "countDocuments", que permite contar o número de documentos que correspondem a uma condição específica.

```
User.countDocuments({ type: 'student' })
   .then(count => { console.log(`There are ${count} students`) })
   .catch(err => { console.log(err) });
```

O Mongoose oferece outra maneira de manipular dados usando os documentos. Podemos usar essa abordagem com:

- Modelos Mongoose, que s\u00e3o classes construtoras que podemos usar para criar novas inst\u00e1ncias do modelo;
- Alguns métodos, como Model.find () em todas as suas formas (.findOne (), .findById ()).

Nestes documentos, podemos usar o método .save () para salvar um documento no banco de dados. Podemos usar .save () para criar um novo documento e atualizar um.

```
----- CREATE ------
// Based on already defined User model, create a user Alice ('Architect')
var myUser = new User({
   name: 'Alice',
   password: 'alice2018',
   job: 'Architect'
});
myUser.save() // Create a new user and return a promise
  .then(user => { console.log('The user was created') })
  .catch(err => { console.log('An error occured', err) });
  ----- UPDATE -----
// Find the user with id '5a3a7ecbc6ca8b9ce68bd41b' and update its job and salary
User.findById('5a3a7ecbc6ca8b9ce68bd41b')
  .then(user => {
   user.job = 'Developer';
   user.salary += 10000;
   return user.save(); // Update the user '5a3a7ecbc6ca8b9ce68bd41b' and return a promise
 1)
  .then(user => { console.log('The user was updated: ' + user )})
  .catch(err => { console.log('An error occured:', err) });
```

Model method	Description
find	Finds documents
findOne	Finds one document
findById	Finds a single document by its _id field
updateMany	Updates many documents
updateOne	Updates one document
findByIdAndUpdate	Updates a single document based on its _id field
deleteMany	Deletes many documents
deleteOne	Deletes one document
findByIdAndRemove	Removes a single document based on its _id field
countDocuments	Counts number of matching documents in a database collection

Document method	Description
save	Saves this document
toObject	Converts this document into a plain javascript object
toString	Helper for console.log

```
us app.js > ...
  const express = require("express")
  const app = express()
  const mongoose = require('mongoose');
  mongoose.connect('mongodb://localhost/ToDoList',
      useNewUrlParser: true,
      useUnifiedTopology: true,
  });
  const index = require("./routes/index")
  const toDoRotas = require("./routes/toDoRoutes")
  app.use(express.json())
  app.use("/", index)
  app.use("/tarefas", toDoRotas)
  module.exports = app
```

```
∨ 📻 models

Js Tarefas.js

{--} tarefas.json
```

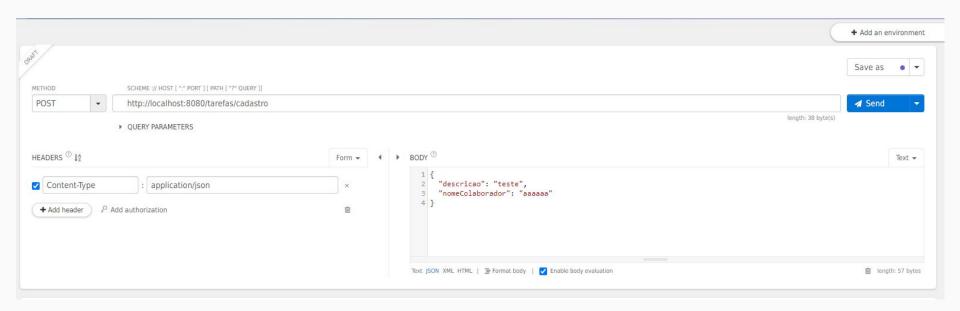
```
src > models > Js Tarefas.js > ...
      const mongoose = require('mongoose');
      const { Schema } = mongoose;
      const userSchema = new Schema({
         concluido: { type: Boolean, default: false },
         descricao: String,
         nomeColaborador: { type: String, required: true },
         },
         { timestamps: true });
 11
       const Tarefas = mongoose.model('Tarefas', userSchema);
 12
 13
 14
      module.exports = Tarefas;
 15
```

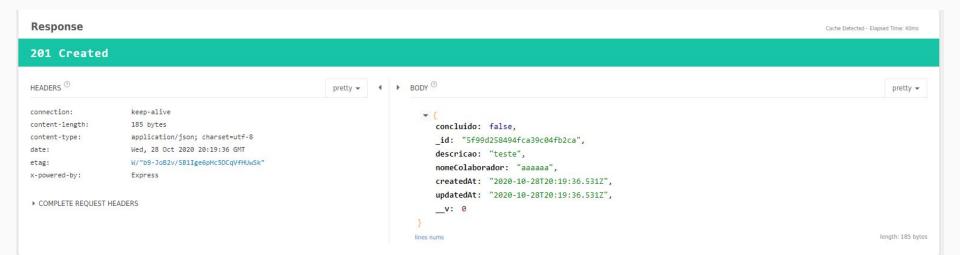
```
const { response, request } = require("express")
const mongoose = require('mongoose');
const Task = require('../models/Tarefas');
```

```
const criarTarefa = (request, response)=>{
   let { descrição, nomeColaborador } = request.body
    /*const novaTarefa ={
        id: Math.random().toString(32).substr(2,9),
       dataInclusao: new Date().toString(),
        concluido: false,
       descricao: descricao,
       nomeColaborador: nomeColaborador
   tarefasModels.push(novaTarefa);
    response.status(201).json(novaTarefa)*/
    /*const novaTarefa ={
       id: Math.random().toString(32).substr(2,9),
       dataInclusao: new Date().toString(),
       concluido: false,
       descricao: descricao,
       nomeColaborador: nomeColaborador
    tarefasModels.push(novaTarefa); */
```

```
const newTask = new Task({
    descricao,
    nomeColaborador,
    });

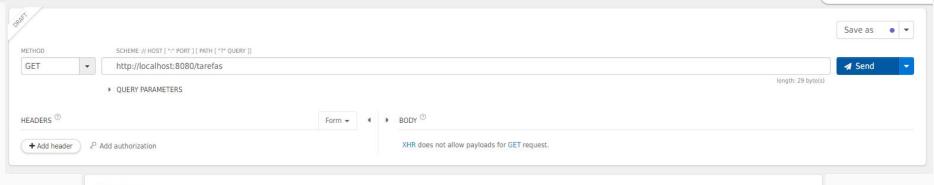
newTask.save()
    .then((res) => {
        response.status(201).json(res);
    })
    .catch(err => next(err));
```

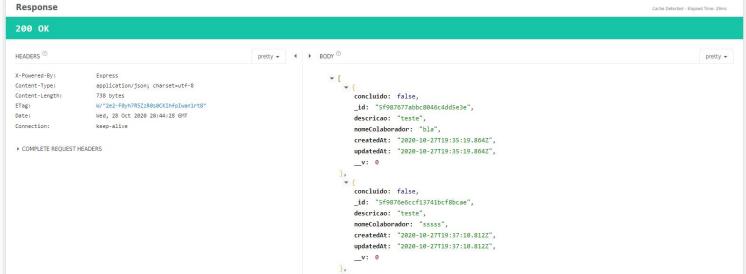




```
const getAll = (request, response)=>{
    // response.status(200).send(tarefasModels)
}
```

```
const getAll = (request, response)=>{
    Task.find()
        .then((tasks) => {
            response.status(200).json(tasks);
        })
        .catch(err => next(err));
}
```





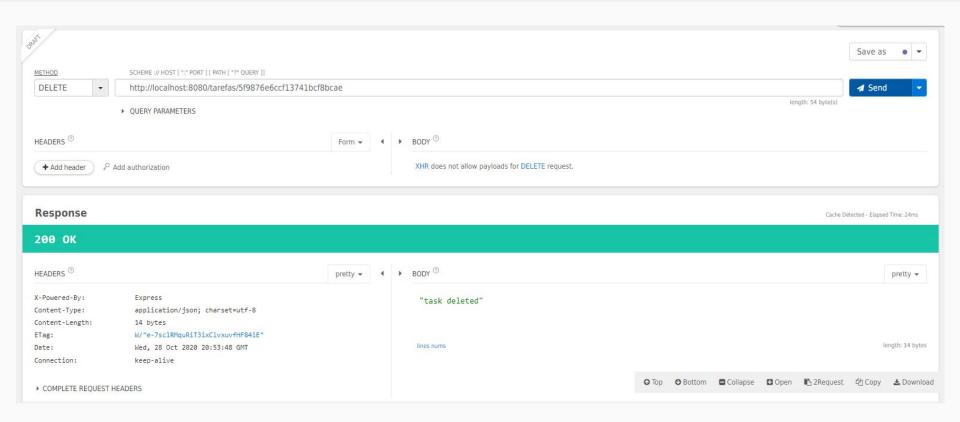
```
const deletarTarefa = (request, response)=>{
   const { id } = request.params
   /*const tarefaFiltrada = tarefasModels.find(tarefa => tarefa.id == id)

   const index = tarefasModels.indexOf(tarefaFiltrada)
    tarefasModels.splice(index, 1)

   response.json({mensagem: "Tarefa deletada com sucesso"}) */
}
```

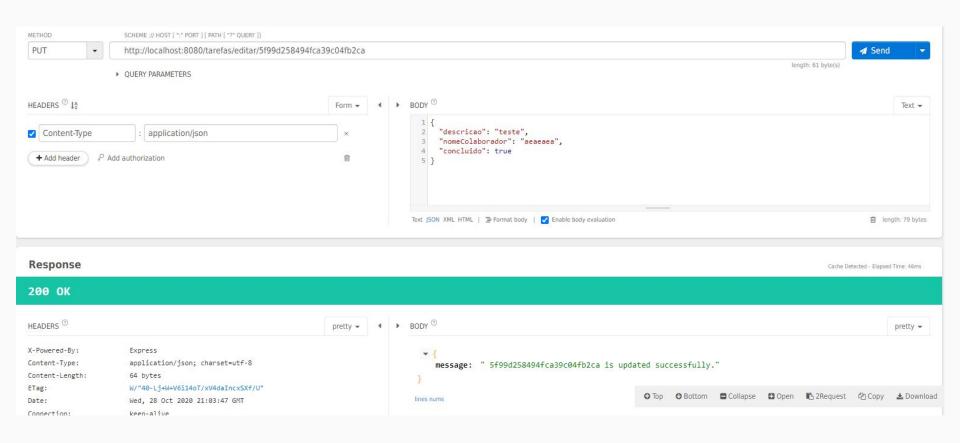
```
const deletarTarefa = (request, response)=>{
  const { id } = request.params

  Task.findByIdAndDelete(id)
    .then(() => {
        response.status(200).json('task deleted');
     })
    .catch((err) => {
        throw new Error(err);
     });
}
```



```
const atualizarTarefa = (request, response) =>{
    const { id } = request.params //pega o ID na URL
    /*const { concluido, descricao, nomeColaborador } = request.body //pega os dados enviados pelo usuário no body
    const tarefaAtualizada = tarefasModels.find(tarefa => tarefa.id == id) //procura a tarefa q será atualizada
    const novaTarefa = { //construir o novo objeto editado
        id: tarefaAtualizada.id, //manter o id que já existe
       dataInclusao: tarefaAtualizada.dataInclusao, //manter a data que já existe
       concluido: concluido, //adicionando o valor "concluido" que foi mandado pelo usuario
       descricao: descricao, //adicionando o valor "descricao" que foi mandado pelo usuario
       nomeColaborador: nomeColaborador //adicionando o valor "nomeColaborador" que foi mandado pelo usuario
   const index = tarefasModels.indexOf(tarefaAtualizada) //procuro a posição dentro do JSON do objeto que será atualizado,
    tarefasModels[index] = novaTarefa //atribuindo a antiga tarefa a nova que construimos
    response.status(200).json(tarefasModels[index])*/
```

```
const atualizarTarefa = (request, response) =>{
    const { id } = request.params //pega o ID na URL
    if (!mongoose.Types.ObjectId.isValid(id)) {
        response.status(400).json({ message: 'Specified id is not valid' });
       return;
    Task.findByIdAndUpdate(id, request.body)
        .then(() => {
            response.status(200).json({ message: ` ${request.params.id} is updated successfully.` });
        .catch((err) => {
           response.json(err);
        });
```



```
/* const tarefa = tarefasModels.find(tarefa => tarefa.id == id)//encontrando a tarefa referente ao ID
  tarefa.concluido = concluido//atualizando o campo "concluido" no nosso JSON
  response.status(200).json({
          mensagem: "Tarefa concluida",
          tarefa
    }) */
```

```
const concluirTarefa = (request, response) => {
    const { id } = request.params //pegando o valor do ID mandado na URL
    const { concluido } = request.body //pegando o valor de "concluido" enviado no Body

Task.findByIdAndUpdate(id, { $set: { concluido }})
    .then((task) => {
        response.status(200).json({ message: `${request.params.id} task is finished.`});
    })
    .catch((err) => {
        response.json(err);
    });
```

