

## Aula 3 - Conectando com o Banco de Dados

### Docupedia Export

Author:Ferro Alisson (CtP/ETS)

Date:17-Aug-2023 15:51

## Table of Contents

Até o momento nós estávamos criando uma API mas os dados ainda não são persistidos. Vamos ver como configurar e conectar o banco de dados MongoDB. Para isso vamos utilizar o mongoose, um ODM de mongoDB para nodeJS, Object Document Mapping (ODM) é um mapeamento para documentos de objetos.

Primeiro vamos criar um arquivo em *config/default.json* e colocar o código

```
{
  "db": "mongodb://localhost:27017/"
}
```

Agora vamos criar um arquivo em *startup/db.js* e colocar o código

```
const mongoose = require('mongoose');
const config = require('config')

module.exports = function(){
  const db = config.get('db');
  mongoose.connect(db, { useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log(`connected to ${db}`));
}
```

E por fim precisamos no arquivo *index.js* importar o *startup/db.js* e executa-lo,

```
const express = require('express');
const app = express();
const routes = require('./routes');

require('./startup/db')();

const port = 8080;

routes(app);

const server = app.listen(port, () => console.log(`Listening on port ${port}`));
```

```
module.exports = server;
```

Assim nosso banco de dados já se conectará com o node,

Agora precisamos criar mais um arquivo em *models/Person.js* vamos criar um model para o banco de dados. Para criar o modelo basta passar como primeiro argumento o nome da *collection*, e as *fields* com os tipos. Como a seguir

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

const Person = mongoose.model('Person', {
  name: String,
  lastname: String,
  salary: Number
})

module.exports = Person;
```

Agora o modelo está pronto, vamos poder utiliza-lo.

Em *routes/person.js* vamos importar o modelo e após isso usar a função create do mongoose.

```
const express = require('express');
const PersonController = require('../controller/PersonController');
const router = express.Router();
const Person = require('../models/Person');
const poeople = [];

router
  .get('/api/person/first', (req, res) => {
    console.log(8+5);
    return
  })
  .get('/api/person', (req, res) => {
    return res.status(200).send({ data: poeople });
  })
  .post('/api/person', async (req, res) => {
    const { name, lastname, salary } = req.body;
```

```
if(!name || !lastname || !salary)
  return res.status(400).send({ message: "Dados inválidos" })

const person = {
  name: name,
  lastname: lastname,
  salary: salary
}

const p = await Person.create(person);
return res.status(201).send({ message: "Pessoa inserida com sucesso", body: p })
})

module.exports = router;
```

Agora nossa rota está pronta para ser utilizada, vamos testá-la no Postman



Mas o que acontece caso o sistema não consiga, por algum motivo, executar a inserção no banco? Vamos testar. Ao fechar o server do banco de dados e executar a requisição novamente, simulando que o banco caiu na hora de inserir os dados. No postman a resposta recebida é a seguinte

## Response



Could not get response

Error: read ECONNRESET | [View in Console](#)

[Learn more about troubleshooting API requests](#)

E no console temos o seguinte resultado

```
U:\node\MongoDB\node_modules\mongodb\lib\sdam\topology.js:277
    const timeoutError = new error_1.MongoServerSelectionError(`Server selection timed out after $
    {serverSelectionTimeoutMS} ms`, this.description);
    ^

MongoServerSelectionError: connect ECONNREFUSED 127.0.0.1:27017
    at Timeout._onTimeout (U:\node\MongoDB\node_modules\mongodb\lib\sdam\topology.js:277:38)
    at listOnTimeout (node:internal/timers:559:17)
    at processTimers (node:internal/timers:502:7) {
  reason: TopologyDescription {
    type: 'Single',
    servers: Map(1) {
      'localhost:27017' => ServerDescription {
        address: 'localhost:27017',
        type: 'Unknown',
        hosts: [],
        passives: [],
        arbiters: [],
        tags: {},
```

```
minWireVersion: 0,
maxWireVersion: 0,
roundTripTime: -1,
lastUpdateTime: 2389064,
lastWriteDate: 0,
error: MongoNetworkError: connect ECONNREFUSED 127.0.0.1:27017
  at connectionFailureError (U:\node\MongoDB\node_modules\mongodb\lib\cmap\connect.js:370:20)
  at Socket.<anonymous> (U:\node\MongoDB\node_modules\mongodb\lib\cmap\connect.js:293:22)
  at Object.onceWrapper (node:events:642:26)
  at Socket.emit (node:events:527:28)
  at emitErrorNT (node:internal/streams/destroy:157:8)
  at emitErrorCloseNT (node:internal/streams/destroy:122:3)
  at processTicksAndRejections (node:internal/process/task_queues:83:21) {
  cause: Error: connect ECONNREFUSED 127.0.0.1:27017
    at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1187:16) {
      errno: -4078,
      code: 'ECONNREFUSED',
      syscall: 'connect',
      address: '127.0.0.1',
      port: 27017
    },
    [Symbol(errorLabels)]: Set(1) { 'ResetPool' }
  },
  topologyVersion: null,
  setName: null,
  setVersion: null,
  electionId: null,
  logicalSessionTimeoutMinutes: null,
  primary: null,
  me: null,
  '$clusterTime': null
}
},
stale: false,
compatible: true,
heartbeatFrequencyMS: 10000,
localThresholdMS: 15,
setName: null,
maxElectionId: null,
```

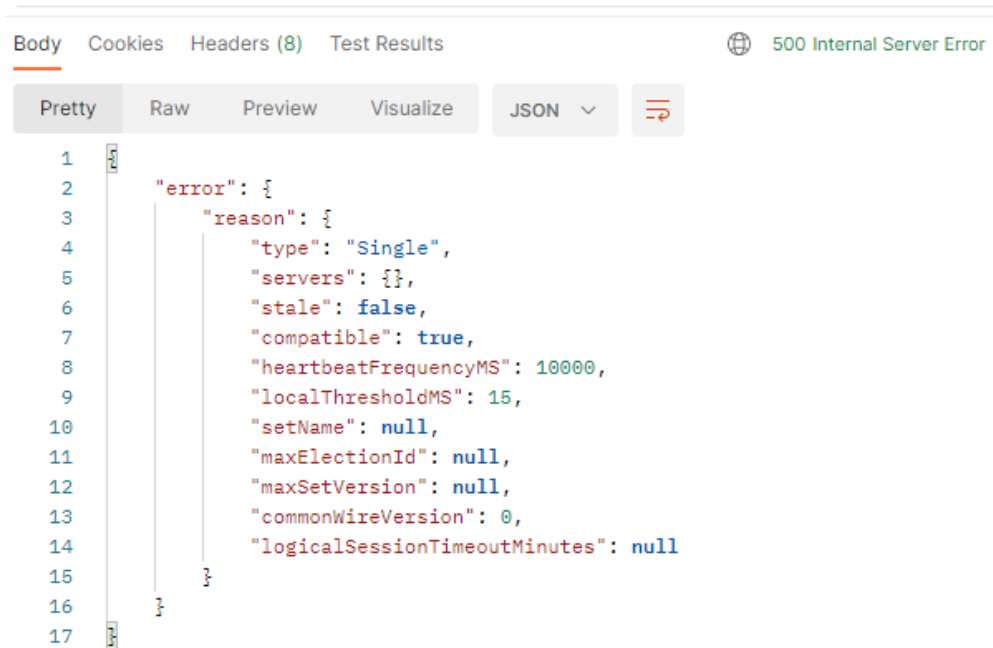
```
maxSetVersion: null,  
commonWireVersion: 0,  
logicalSessionTimeoutMinutes: null  
},  
code: undefined,  
[Symbol(errorLabels)]: Set(0) {}  
}  
[nodemon] app crashed - waiting for file changes before starting...
```

A API caiu, e isso não é uma boa prática, visto que uma execução simples como adicionar um elemento no banco, mas com o banco indisponível derrubou toda a aplicação, para prevenir isso vamos colocar um *try catch* no nosso código.

```
.post('/api/person', async (req, res) => {  
  const { name, lastname, salary } = req.body;  
  if(!name || !lastname || !salary)  
    return res.status(400).send({ message: "Dados inválidos" });  
  
  const person = {  
    name: name,  
    lastname: lastname,  
    salary: salary  
  }  
  
  try {  
    const p = await Person.create(person);  
    return res.status(201).send({ message: "Pessoa inserida com sucesso", body: p });  
  } catch (error) {  
    return res.status(500).send({ error: error });  
  }  
})
```

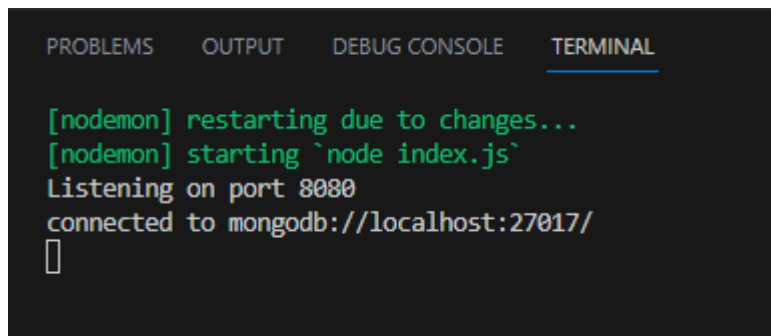
Agora vamos testar da mesma forma, após a API rodando fechar o server do banco de dados e enviar a requisição no Postman.





```
Body Cookies Headers (8) Test Results 500 Internal Server Error
Pretty Raw Preview Visualize JSON
1
2   "error": {
3     "reason": {
4       "type": "Single",
5       "servers": {},
6       "stale": false,
7       "compatible": true,
8       "heartbeatFrequencyMS": 10000,
9       "localThresholdMS": 15,
10      "setName": null,
11      "maxElectionId": null,
12      "maxSetVersion": null,
13      "commonWireVersion": 0,
14      "logicalSessionTimeoutMinutes": null
15    }
16  }
17
```

E no console



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Listening on port 8080
connected to mongodb://localhost:27017/
█
```

Ou seja, recebemos um código de erro no Postman, mas a aplicação não caiu.

Agora vamos na requisição *GET* para alterar para buscar no banco de dados.  
Para requisições *GET* usaremos o método `Model.find()` e da mesma forma o *try catch*

```
.get('/api/person', async (req, res) => {  
  try {  
    const people = await Person.find();  
    return res.status(200).send({ data: people });  
  } catch (error) {  
    return res.status(500).send({ error: error });  
  }  
})
```

E vamos testar no Postman



Como só tínhamos um dado cadastrado, só nos retornou um dado, vamos popular o banco para que podemos ver melhor o retorno. Agora com mais de um dado, vamos criar um método para buscar por um id específico.

```
.get('/api/person/:id', async (req, res) => {  
  const { id } = req.params;  
  
  try {  
    const person = await Person.findById(id);
```

```
    return res.status(200).json(person);
  } catch (error) {
    res.status(500).json({ error: error })
  }
})
```



Na aula anterior, foi trabalhado com as requisições de criação e busca de valores, nessa aula veremos como fazer requisições para alterar e deletar dados. Para alterar dados usaremos o verbo *HTTP PATCH* como visto na aula 2 sobre verbos *HTTP*. Da mesma forma que as outras requisições, o primeiro argumento é uma string com o endpoint e o segundo argumento é uma função *callback assíncrona*.

```
.patch('/api/person/:id', async (req, res) => {
  const { id } = req.params;
  if(!id)
    return res.status(400).send({ message: "No id provider" })

  const person = req.body;
  if(!person.salary)
    return res.status(400).send({ message: "No salary provider" })

  try {
    const newPerson = await Person.findByIdAndUpdate(
      id,
```

```
        { salary: person.salary }
    );
    return res.status(201).send(newPerson);
} catch (error) {
    return res.status(500).send({ error: error });
}
})
```

e vamos ao Postman para testar a API.



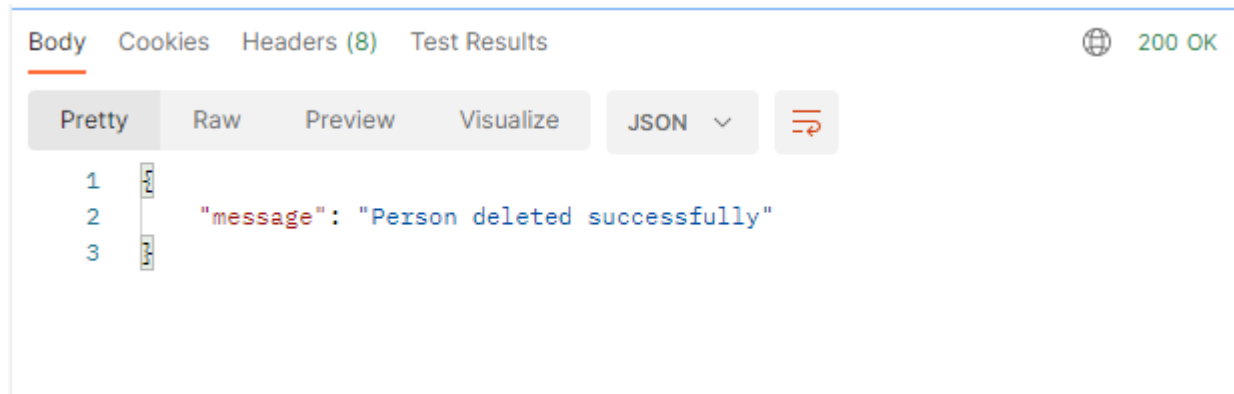
Para deletar é de forma semelhante, o primeiro argumento é uma string com o endpoint e o segundo argumento é uma função *callback assíncrona*.

```
.delete('/api/person/:id', async (req, res) => {
    const { id } = req.params;
    if(!id)
        return res.status(400).send({ message: "No id provider" });

    try {
        await Person.findByIdAndRemove(id);
        return res.status(200).send({ message: "Person deleted successfully" })
    } catch (error) {
```

```
    console.log(error);  
    return res.status(500).send({ message: "Something failed"})  
  }  
})
```

E no Postman vamos testar a requisição



Desafio: Crie uma API com banco de dados para produtos esportivos, inicialmente somente a *collection* de produtos.