

UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO  
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUIS FERNANDO PEIXOTO CABRAL  
RALF CRUZ MATEUS  
YURI GARCIA CAMPOS

**TRABALHO PRÁTICO – FASE FINAL**

CAMPOS DOS GOYTACAZES – RJ  
2021

## SUMÁRIO

<b>1. INTRODUÇÃO .....</b>	<b>2</b>
1.1    Objetivo .....	2
<b>2. TECNOLOGIAS E BASE DE DADOS.....</b>	<b>2</b>
2.1    2.1 Rotas da Aplicação .....	3
2.2    2.2 Banco de dados escolhido .....	3
2.3    2.3 Fonte de dados .....	4
<b>3. MODELAGEM DOS DADOS.....</b>	<b>4</b>
3.1    3.1 Explicação do modelo .....	5
3.2    3.2 Detalhes dos campos do modelo .....	5
3.2.1    3.2.1 Collection usuario .....	5
3.2.2    3.2.2 Collection conversa .....	5
3.2.3    3.2.3 Collection mensagem .....	6
<b>4. INSTALAÇÃO E CONFIGURAÇÃO DO BANCO DE DADOS .....</b>	<b>6</b>
4.1    Criação do Cluste no Atlas .....	6
4.2    Conexão com o banco de dados.....	8
<b>5. EXPLICAÇÃO DA APLICAÇÃO .....</b>	<b>9</b>
5.1    Modelos e Esquemas .....	10
5.2    Controllers da aplicação .....	13
5.2.1    Controller de autenticação.....	13
5.2.1.1    Login .....	13
5.2.1.2    Register.....	14
5.2.2    Controller de usuario .....	16
5.2.2.1    Get .....	16
5.2.2.2    Update.....	17
5.2.2.3    Followers.....	18
5.2.2.4    Follow .....	19
5.2.2.5    Unfollow .....	20
5.2.3    Controller de Chat.....	20
5.2.3.1    Get .....	20
5.2.3.2    NewChat .....	22
5.2.4    Controller das mensagens.....	23
5.2.4.1    Get .....	24

5.2.4.2	NewMessage .....	25
<b>6.</b>	<b><i>IMAGENS DAS COLLECTIONS CRIADAS</i></b> .....	<b>27</b>
<b>7.</b>	<b><i>TESTE DE REQUISIÇÕES</i></b> .....	<b>28</b>
<b>7.1</b>	<b>Métricas</b> .....	<b>28</b>
<b>7.2</b>	<b>Ambientes testados</b> .....	<b>28</b>
<b>7.3</b>	<b>Cenários</b> .....	<b>29</b>
7.3.1	Requisições assíncronas.....	29
7.3.1.1	Um usuário realiza o envio de requisições de escrita .....	29
7.3.1.2	Dois usuários realizam o envio de requisições de escrita simultâneas .....	30
7.3.2	Requisições síncronas.....	30
7.3.2.1	Um usuário realiza o envio de requisições de leitura .....	31
7.3.2.2	Um usuário realiza o envio de requisições de escrita .....	31
7.3.2.3	Dois usuários realizam o envio de requisições de leitura e escrita .....	31
<b>7.4</b>	<b>Realização dos testes</b> .....	<b>32</b>
7.4.1	Teste de leitura .....	33
7.4.2	Teste de escrita .....	33
<b>7.5</b>	<b>Resultados</b> .....	<b>34</b>
7.5.1	Requisições assíncronas.....	34
7.5.1.1	Um usuário realiza o envio de requisições de escrita .....	34
7.5.1.2	Dois usuários realizam o envio de requisições de escrita e leitura .....	36
7.5.2	Requisições síncronas.....	39
7.5.2.1	Um usuário realiza o envio de requisições de escrita .....	39
7.5.2.2	Um usuário realiza o envio de requisições de leitura .....	40
7.5.2.3	Dois usuários realizam o envio de requisições de leitura e escrita .....	41
<b>8.</b>	<b>CONCLUSÃO</b> .....	<b>42</b>
<b>9.</b>	<b>CRONOGRAMA DO PROJETO</b> .....	<b>43</b>
<b>10.</b>	<b><i>REPOSITÓRIO DA APLICAÇÃO NO GITHUB</i></b> .....	<b>43</b>
<b>11.</b>	<b>BIBLIOGRAFIA</b> .....	<b>43</b>

## 1. INTRODUÇÃO

Este relatório tem como objetivo apresentar a fase intermediaria do trabalho prático da disciplina de Processamento Massivo de Dados.

O tema escolhido para o projeto foi a Aplicação de Mensageria: aplicação simples usando um NoSQL para armazenamento + Teste de requisições.

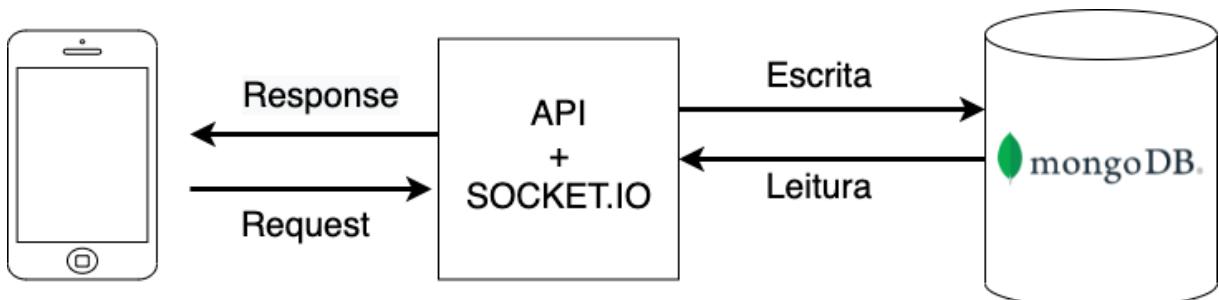
### 1.1 Objetivo

O desenvolvimento de um aplicativo mobile, intitulado como "Campion", que se trata de um aplicativo de troca de mensagens, onde um usuário pode mandar e receber mensagens facilmente com os demais usuários do aplicativo. Também será desenvolvido uma API REST para ser consumida pelo aplicativo.

## 2. TECNOLOGIAS E BASE DE DADOS

Será utilizada para o desenvolvimento da API a linguagem JavaScript, o ambiente de desenvolvimento NodeJS e o framework Express. Será utilizado no aplicativo mobile a biblioteca Axios para consumir a API.

Para o realizar a comunicação em tempo real entre os usuários da aplicação será utilizado o socket.io, que se trata de uma biblioteca em que permite a comunicação entre clientes e servidores.



**Figura 1:** Arquitetura da aplicação

### **2.1 2.1 Rotas da Aplicação**

Será desenvolvido para a aplicação as seguintes rotas:

<b>Rotas</b>	<b>Métodos</b>	<b>Função</b>
<b>AUTENTICAÇÃO</b>		
auth/login	POST	Realiza o login na aplicação
auth/register	POST	Cadastra um novo usuário
<b>CONVERSA</b>		
chats/:userId	GET	Retorna a todas as conversas de um usuário
chats/	POST	Criar uma nova conversa
chats/find/:userId/:secondUserId	GET	Retorna a conversa entre dois usuários
<b>MENSAGEM</b>		
message/:chatId	GET	Retorna todas as mensagens de uma conversa
message/	POST	Criar uma nova mensagem
<b>USUÁRIO</b>		
user/	GET	Retorna todos os usuários
user/:userId	PUT	Atualiza os dados cadastrais de um usuário
user/followers/:userId	GET	Retorna todos os seguidores de um usuário
user/:userId/follow	PUT	Segui um usuário
user/:userId/unfollow	PUT	Deixa de seguir um usuário

### **2.2 2.2 Banco de dados escolhido**

O sistema de gerenciamento de banco de dados escolhido foi o MongoDB por ser mais familiar. A princípio optamos por utilizar o MongoDB na nuvem a versão gratuita, porém caso seja encontrada alguma limitação ao realizar os testes de

requisições, iremos realizar a trocar pelo MongoDB local. No entanto, caso seja feita a troca iremos adicionar no relatório final as limitações que encontrarmos e os teste realizados no MongoDB atlas.

### 2.3 2.3 Fonte de dados

Como a aplicação de mensageria gera seus próprios dados não estaremos especificando aqui uma fonte de dados.

## 3. MODELAGEM DOS DADOS

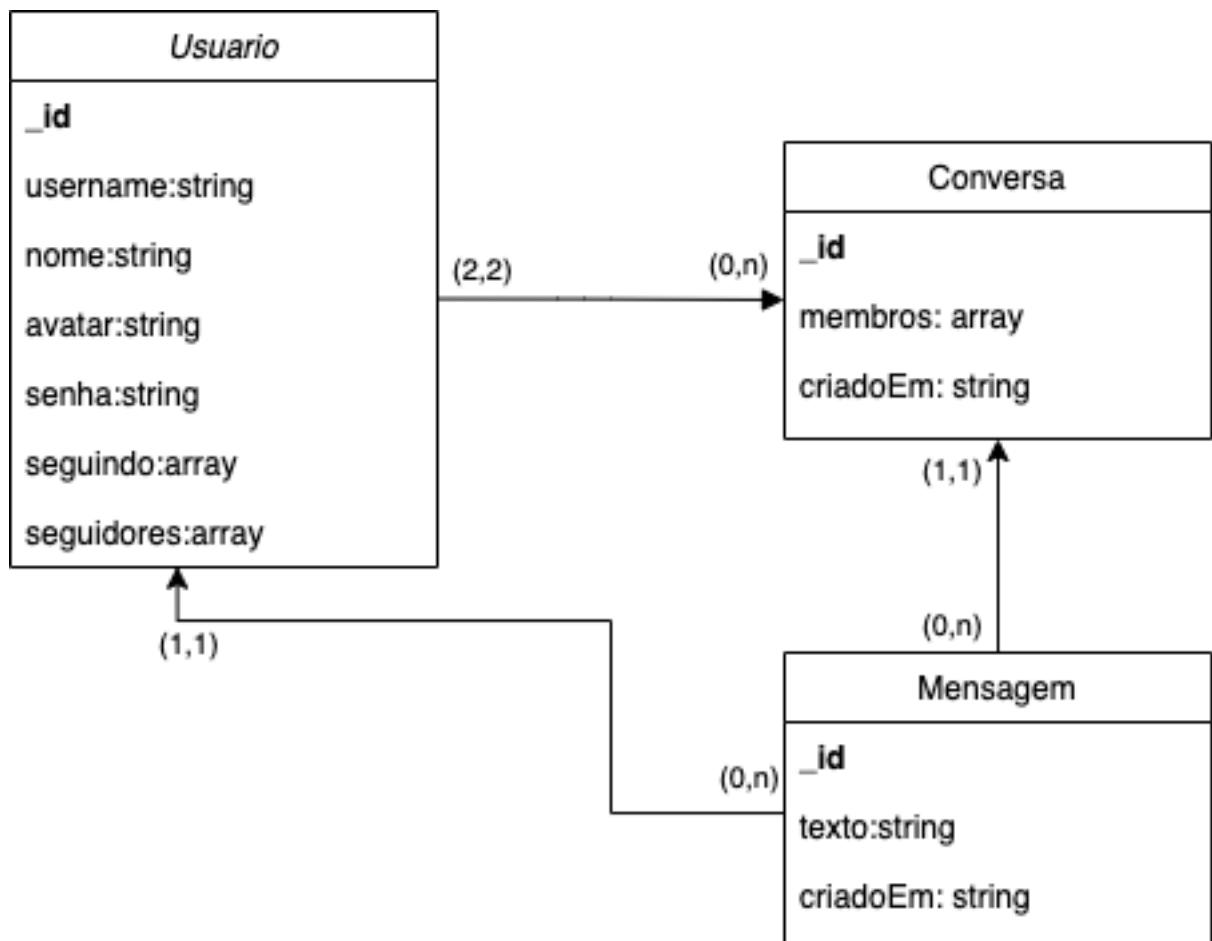


Figura 2 : Modelo de dados

### **3.1 3.1    Explicação do modelo**

O modelo da aplicação é composto por três entidades: usuário, conversa e mensagem. Cada usuário é identificado por seu ID e pode iniciar uma conversa com qualquer outro usuário que esteja tanto na lista de seguidores quanto na de seguindo. A conversa vai ter somente 2 usuários participantes enquanto cada usuário pode estar em n diferentes conversas. Pertence a uma conversa os dois usuários que estão com o seus Ids em membros. A mensagem pode ser enviada n vezes por cada usuário.

### **3.2 3.2    Detalhes dos campos do modelo**

Serão criadas as três collections, usuario, conversa e mensagem para ser armazenados no banco de dados. Temos os seguintes campos em cada uma das coleções:

#### **3.2.1 3.2.1    Collection usuario**

- \_id:** Identificador que será gerado pelo mongoDB
- username:** Identificador único para cada usuário, será utilizado para que os usuários dentro da aplicação possam encontrar outros usuários.
- nome:** Nome completo do usuário
- avatar:** Endereço da imagem do perfil
- senha:** Senha criptografada do usuário
- seguindo:** Array contendo todos \_id de outros usuários que são seguidos
- seguidores:** Array contendo todos \_id dos outros usuários que seguem o usuário

#### **3.2.2 3.2.2    Collection conversa**

- \_id:** Identificador que será gerado pelo mongoDB
- membros:** Array com dois \_id de usuarios que fazem parte da conversa
- criadoEm:** Hora e data do inicio da conversa

### **3.2.3 Collection mensagem**

**\_id:** Identificador que será gerado pelo mongoDB

**texto:** Texto da mensagem

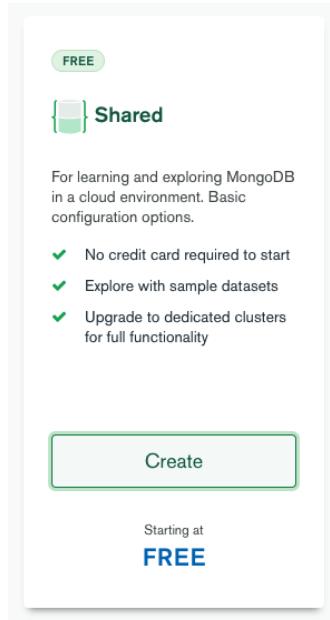
**criadoEm:** Hora e data do envio da mensagem

## **4. INSTALAÇÃO E CONFIGURAÇÃO DO BANCO DE DADOS**

Definimos o MongoDB ATLAS como o sistema de gerenciamento de banco de dados da aplicação por ser familiar para os integrantes do grupo.

### **4.1 Criação do Cluste no Atlas**

Primeiramente escolhemos o plano gratuito:



Em seguida definimos o Google Cloud como servidor cloud e escolhemos a São Paulo como a região. O cluster possui um tamanho de armazenamento de 512MB e a versão do MongoDB é a 4.4.

**Cloud Provider & Region**

GCP, Sao Paulo (southamerica-east1) ▾

★ Recommended region ⓘ ⚡ Paid tier region ⓘ

NORTH AMERICA / SOUTH AMERICA	EUROPE / MIDDLE EAST / AFRICA	ASIA PACIFIC
Sao Paulo (southamerica-east1) ★	Belgium (europe-west1) ★	Singapore (asia-southeast1) ★
Iowa (us-central1) ★	Netherlands (europe-west4) ★	Taiwan (asia-east1) ★
Los Angeles (us-west2) ★ ⓘ	Finland (europe-north1) ★ ⓘ	Mumbai (asia-south1) ★
Las Vegas (us-west4) ★ ⓘ	Zurich (europe-west6) ★ ⓘ	Tokyo (asia-northeast1) ★
N. Virginia (us-east4) ★ ⓘ	Frankfurt (europe-west3) ★ ⓘ	Osaka (asia-northeast2) ★ ⓘ
South Carolina (us-east1) ★ ⓘ	London (europe-west2) ★ ⓘ	Hong Kong (asia-east2) ★ ⓘ
Oregon (us-west1) ★ ⓘ	AUSTRALIA	Jakarta (asia-southeast2) ★ ⓘ
Salt Lake City (us-west3) ★ ⓘ	Sydney (australia-southeast1)	Seoul (asia-northeast3) ★ ⓘ
Montreal (northamerica-northeast1) ★ ⓘ	★ ⓘ	

**Cluster Tier**

M0 Sandbox (Shared RAM, 512 MB Storage)  
Encrypted ▾

**Additional Settings**

MongoDB 4.4, No Backup ▾

**Cluster Name**

Cluster0 ▾

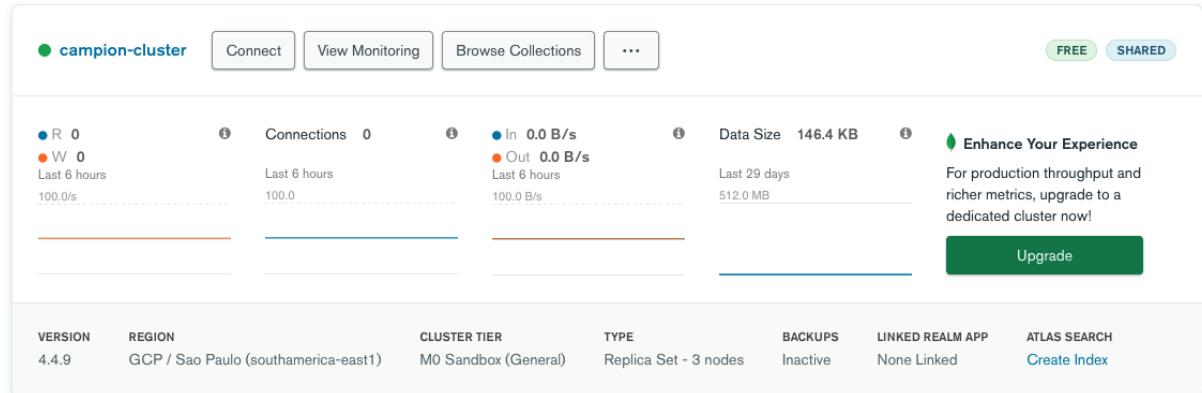
**FREE**

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

[Back](#)

[Create Cluster](#)

Com isso foi criado o cluster que será utilizado na aplicação.



## 4.2 Conexão com o banco de dados

Para realiza a conexão com o banco de dados utilizamos a opção de “Connect your application”, onde foi selecionado o NodeJs na versão 4.0 ou superior. Com isso foi gerado uma URL para realizar a conexão com a aplicação.

Connect to campion-cluster

✓ Setup connection security > ✓ Choose a connection method > Connect

**1 Select your driver and version**

DRIVER	VERSION
Node.js	4.0 or later

**2 Add your connection string into your application code**

Include full driver code example

```
mongodb+srv://campion_luispeixoto:<password>@campion-cluster.lsyp3.mongodb.net/myFirstDatabase?retryWrites=true&w=majority
```

Replace **<password>** with the password for the **campion\_luispeixoto** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are **URL encoded**.

Having trouble connecting? [View our troubleshooting documentation](#)

[Go Back](#)

[Close](#)

Foi necessário instalar a biblioteca mongoose na versão 6.0, para realizar a conexão com o banco de dados.

A partir disso foi realizado a conexão utilizando a função connect.

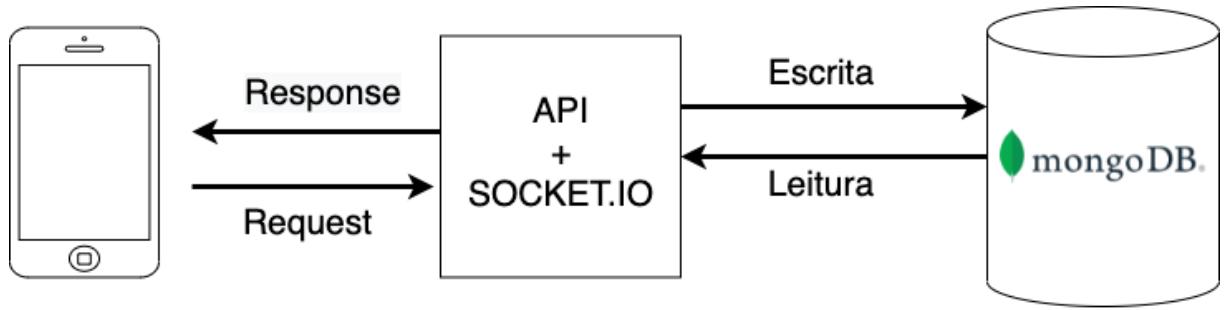
```
src > database > index.js > ...
1  const mongoose = ...
2
3  const URI = process.env.MONGO_DB
4
5  const connectDB = async () => {
6    await mongoose.connect(URI, {
7      useUnifiedTopology: true
8    }, () => {
9      console.log('Connected DB')
10   })
11 }
12
13 module.exports = connectDB
14
```

Rodando a aplicação, temos que foi feita a conexão.

```
~/projects/campion-server Final ↓
→ sudo yarn start
Password:
yarn run v1.22.11
$ nodemon src/server.js
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/server.js`
Connected DB
□
```

## 5. EXPLICAÇÃO DA APLICAÇÃO

A aplicação foi desenvolvida utilizando a linguagem JavaScript, o ambiente de desenvolvimento NodeJS e o framework Express. Para realização a comunicação em tempo real entre os usuários da aplicação foi utilizado o socket.io.



**Figura:** Arquitetura da aplicação

## 5.1 Modelos e Esquemas

Para o chat foi definido o esquema de members, que é do tipo Array, que deve receber os Ids dos usuários pertencentes a uma conversa. Foi passado o timestamps como true, onde essa configuração permite que quando for criado uma nova conversa será gravado a data e a hora no registro.

```

src > models > Chat.js > ...
1  const mongoose = ...
2
3  const { Schema } = mongoose
4
5  const chatSchema = new Schema({
6      members: {
7          type: Array
8      }
9  },
10 { timestamps: true }
11 )
12
13 module.exports = mongoose.model('chat', chatSchema)
14 |

```

O esquema do message é definido pelo chatId: que é do tipo string, é onde receber os ID do chat da qual aquela mensagem pertence; sender é do tipo string e receber o id do usuário remetente da mensagem; receiver é do tipo string e recebe o id do usuário destinatário; text é do tipo string e receber o texto da mensagem; Por fim é passado o timestamps como verdadeiro.

```
src > models > Message.js > ...
1  const mongoose = require('mongoose')
2
3  const { Schema } = mongoose
4
5  const messageSchema = new Schema({
6    chatId: {
7      type: String
8    },
9
10   sender: {
11     type: String
12   },
13   receiver: {
14     type: String
15   },
16   text: {
17     type: String
18   }
19 },
20 {
21   timestamps: true
22 })
23
24 module.exports = mongoose.model('message', messageSchema)
25 |
```

O esquema do user é definido pelo: username, que é obrigatório, único, tem que ter pelo menos três caracteres e é do tipo string; name é do tipo string, é obrigatório e tem que ter pelo menos 3 caracteres; password é um campo obrigatório, tem que ter pelo menos 8 caracteres e é um campo obrigatório; avatar é um campo do tipo string e é onde deve receber o endereço de imagens; followers é do tipo array e receber os IDs dos usuários na qual são seguidores; followings é o campo do tipo array e recebe os IDs dos usuários que estão sendo seguidos.

```
src > models > User.js > ...
1  const mongoose = require('mongoose')
2
3  const { Schema } = mongoose
4
5  const userSchema = new Schema({
6    username: {
7      type: String,
8      required: true,
9      min: 3,
10     unique: true
11   },
12
13   name: {
14     type: String,
15     required: true,
16     min: 3
17   },
18
19   password: {
20     type: String,
21     required: true,
22     min: 8
23   },
24
25   avatar: {
26     type: String,
27     default: ''
28   },
29
30   followers: {
31     type: Array,
32     default: []
33   },
34
35   followings: {
36     type: Array,
37     default: []
38   }
39
40 })
41
42 module.exports = mongoose.model('user', userSchema)
43 |
```

## 5.2 Controllers da aplicação

Quando é acessado uma rota, os controllers realizam o tratamento dos dados recebidos retorna uma resposta.

### 5.2.1 Controller de autenticação

No controller de autenticação possui duas funções a de login e de register.

#### 5.2.1.1 Login

A função de login receber o username e password e verifica se possui um usuário no banco de dados com o username recebido, verifica se a senha está certa e por fim retorna os dados do usuario e token.

```
src > controllers > authController.js > [o] <unknown>
  7  async login (req, res) {
  8    try {
  9      const { username, password } = req.body
 10      const user = await User.findOne({ username })
 11
 12      if (!user) {
 13        return res.status(404).json({ message: 'error' })
 14      }
 15
 16      const isValidPassword = await validatePassword(password, user.password)
 17      if (!isValidPassword) {
 18        return res.status(404).json({ message: 'error' })
 19      }
 20
 21      /// CRIAR O TOKEN PARA O USUARIO
 22
 23      const { secret, expiresIn } = authConfig.jwt
 24
 25      const token = sign({}, secret, {
 26        subject: user._id.toString(),
 27        expiresIn
 28      })
 29
 30      const data = {
 31        _id: user._id.toString(),
 32        username: user.username,
 33        name: user.name,
 34        avatar: user.avatar,
 35        followers: user.followers,
 36        followings: user.followings
 37
 38    }
```

```

39         return res.status(200).json({ user: data, token })
40     } catch (error) {
41         console.log(error)
42         if (error) { return res.status(500).send({ error: error }) }
43     }
44 }
45 ,

```

Dados recebidos:

POST ➔ http://localhost:3000/auth/login

JSON ▾

```

1▼ {
2   "username": "luispeixoto",
3   "password": "123456789"
4 }

```

Retorno:

200 OK    127 ms    334 B

Preview ▾

Header 16    Cookie    Timeline

```

1▼ {
2   "user": {
3     "_id": "61364da2c24be611d2f05b2f",
4     "username": "luispeixoto",
5     "name": "Luis",
6     "avatar": "",
7     "followers": [
8       "613677d2f750078ce68142aa"
9     ],
10    "followings": []
11  },
12  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE2MzMxMzgwNjksImV4cCI6MTYzMzc0Mjg2OSwiOJmuTTY9VciDXDh5gqLKcz6VtBF5mfqFU"
13 }

```

### 5.2.1.2 Register

A função register é responsável por criar um novo usuário, portanto a função recebe como entrada o username, name e o password. É feita uma verificação no banco de dados se existe um outro usuário com o username recebido como entrada, caso não tenha nenhum registro do username no banco de dados, é criado um novo registro de usuário com os dados recebidos.

```

47  async register (req, res) {
48    try {
49      const { username, name, password } = req.body
50      const checkUsername = await User.findOne({ username })
51
52      if (checkUsername) {
53        return res.status(409).json({ message: 'username is already in use' })
54      }
55
56      const hashedPassword = await hashPassword(password)
57
58      const user = new User({
59        username,
60        name,
61        password: hashedPassword
62      )
63
64      const data = user.save()
65      res.status(200).json(data)
66    } catch (error) {
67      console.log(error)
68      if (error) { return res.status(500).send({ error: error }) }
69    }
70  }
71
72}
73

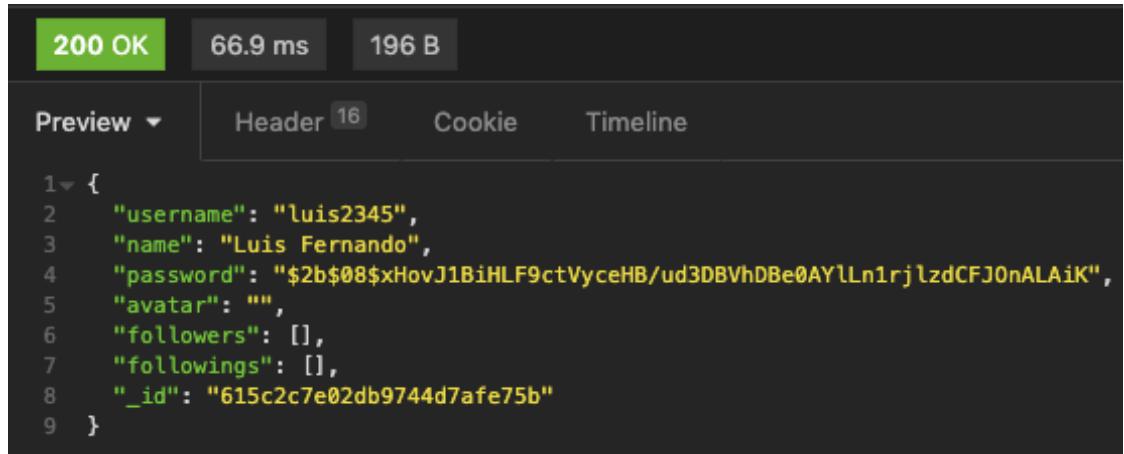
```

Dados recebidos:

POST ▾ http://localhost:3000/auth/register

JSON	Auth	Query	Header 1
<pre> 1 {  2   "username": "luis2345",  3   "name": "Luis Fernando",  4   "password": "123456789"  5 }</pre>			

Retorno:



200 OK    66.9 ms    196 B

Preview ▾ Header 16 Cookie Timeline

```
1< {  
2   "username": "luis2345",  
3   "name": "Luis Fernando",  
4   "password": "$2b$08$xHovJ1BiHLF9ctVyceHB/ud3DBVhDBe0AYlLn1rjLzdCFJ0nALAiK",  
5   "avatar": "",  
6   "followers": [],  
7   "followings": [],  
8   "_id": "615c2c7e02db9744d7afe75b"  
9 }
```

### 5.2.2 Controller de usuário

O controller de usuário possui as funções de get, update, followers, follow, unfollow.

#### 5.2.2.1 Get

A função Get é responsável por retorna os dados do usuário de acordo com o username de entrada. Onde é feito uma busca no banco de dados por usuário que possui o username recebido. Caso seja encontrado um registro, é retornado o resultado sem os campos password e updatedAt.

```
5  async get (req, res) {  
6    const { username } = req.query  
7    try {  
8      const user = await User.findOne({ username })  
9      const { password, updatedAt, ...other } = user._doc  
10     res.status(200).json(other)  
11    } catch (error) {  
12      console.log(error)  
13      if (error) { return res.status(500).send({ error: error }) }  
14    }  
15  },
```

Dados recebidos:

GET ▾ http://localhost:3000/user/

Multipart 1 ▾ Auth ▾ Query 1 Header 1 Docs

URL PREVIEW  
http://localhost:3000/user/?username=luispeixoto

☰ username luispeixoto

Retorno:

200 OK 29.2 ms 176 B

Preview ▾ Header 16 Cookie Timeline

```
1▼ {  
2   "_id": "61364da2c24be611d2f05b2f",  
3   "username": "luispeixoto",  
4   "name": "Luis",  
5   "avatar": "",  
6   "followers": [  
7     "613677d2f750078ce68142aa"  
8   ],  
9   "followings": [  
10    "613677d2f750078ce68142aa"  
11  ],  
12  "__v": 0  
13 }
```

### 5.2.2.2 Update

A função realiza a atualização das informações do usuário no banco de dados

```

17  async update (req, res) { // Atualiza as informações do usuário
18    const { id } = req.params
19
20    if (req.body.userId === req.params.id) {
21      try {
22        if (req.body.password) {
23          req.body.password = await hashPassword(req.body.password)
24        }
25
26        const user = await User.findByIdAndUpdate(id, {
27          $set: req.body
28        })
29
30        user.save()
31        res.status(200).json({ message: 'updated' })
32      } catch (error) {
33        console.log(error)
34        if (error) { return res.status(500).send({ error: error }) }
35      }
36    }
37  },

```

### 5.2.2.3 Followers

A função followers retorna uma lista de seguidores de um determinado usuário de acordo com o ID de entrada. É feita uma busca pelo ID do usuário no banco de dados, onde através do registro retornado, é pego todos os IDs do campo followings e para cada ID é feito uma busca no banco de dados pegando as informações daquele usuário. Onde no final é retornado uma lista de usuários que seguem aquele usuário do ID de entrada.

```

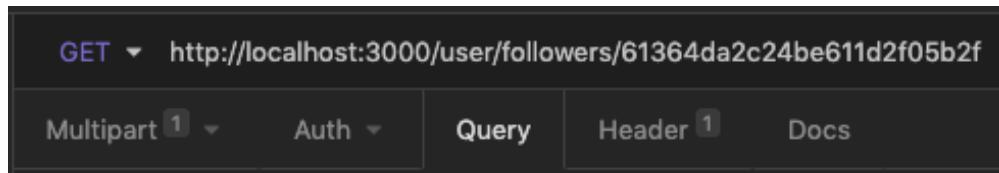
async followers (req, res) { // retorna todos os seguidores de um usuário
  try {
    const user = await User.findById(req.params.userId)
    const followings = await Promise.all(
      user.followings.map((followerId) => {
        return User.findById(followerId)
      })
    )

    const followersList = []

    followings.map((follower) => {
      const { _id, username, name, avatar } = follower
      followersList.push({ _id, username, name, avatar })
    })
    res.status(200).json(followersList)
  } catch (error) {
    console.log(error)
    if (error) { return res.status(500).send({ error: error }) }
  }
},

```

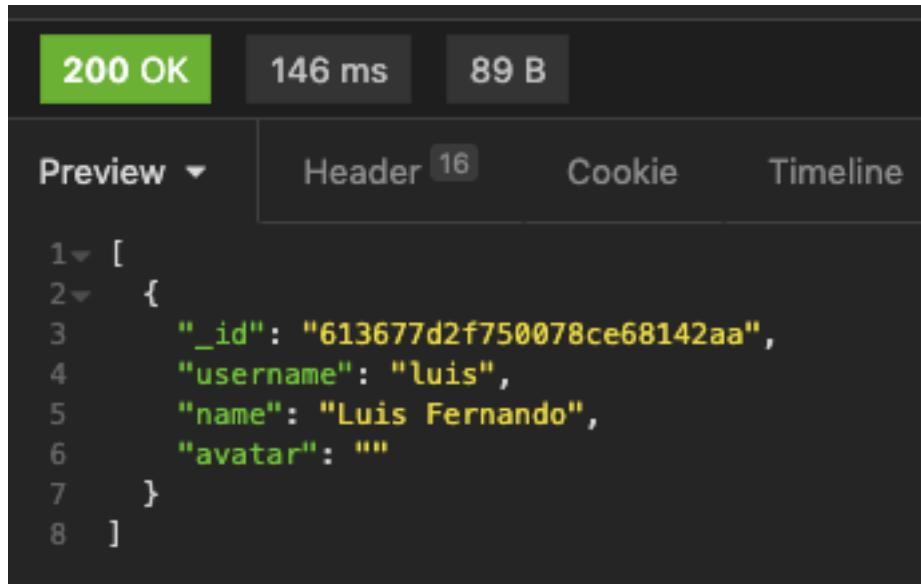
Dados recebidos:



GET ▾ http://localhost:3000/user/followers/61364da2c24be611d2f05b2f

Multipart 1 ▾ Auth ▾ Query Header 1 Docs

Retorno:



200 OK 146 ms 89 B

Preview ▾ Header 16 Cookie Timeline

```
1 [ 
2   { 
3     "_id": "613677d2f750078ce68142aa",
4     "username": "luis",
5     "name": "Luis Fernando",
6     "avatar": ""
7   }
8 ]
```

#### 5.2.2.4 Follow

Follow é responsável por colocar no campo followers e followings o IDs dos usuários.

```
61 | async follow (req, res) { // seguir um outro um usuario atraves de um id
62 |   const { id } = req.params
63 |   if (req.body.userId !== id) {
64 |     try {
65 |       const user = await User.findById(id)
66 |       const currentUser = await User.findById(req.body.userId)
67 |
68 |       if (!user.followers.includes(req.body.userId)) {
69 |         await user.updateOne({ $push: { followers: req.body.userId } })
70 |         await currentUser.updateOne({ $push: { followings: id } })
71 |       }
72 |       res.status(200).json('followed')
73 |     } catch (error) {
74 |       console.log(error)
75 |       if (error) { return res.status(500).send({ error: error }) }
76 |     }
77 |   }
78 | },
79 | }
```

### 5.2.2.5 Unfollow

Função que remove dos campos followers e followings o IDs dos usuários.

```
80  async unfollow (req, res) { // deixa de seguir um usuario atraves do id
81    const { id } = req.params
82    if (req.body.userId !== id) {
83      try {
84        const user = await User.findById(id)
85        const currentUser = await User.findById(req.body.userId)
86
87        if (user.followers.includes(req.body.userId)) {
88          await user.updateOne({ $pull: { followers: req.body.userId } })
89          await currentUser.updateOne({ $pull: { followings: id } })
90        }
91        res.status(200).json('Unfollowed')
92      } catch (error) {
93        console.log(error)
94        if (error) { return res.status(500).send({ error: error }) }
95      }
96    }
97  }
98
99 }
```

### 5.2.3 Controller de Chat

O chat possui as funções de Get e NewChat

#### 5.2.3.1 Get

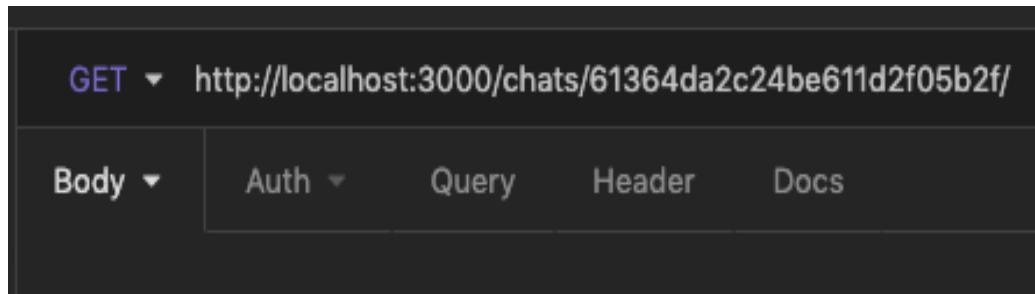
A função Get é responsável por retorna todas as conversas de um usuário, onde recebe como entrada o ID do usuário, sendo assim é realizado uma busca do ID no banco de dados collection chats no campo members. Em seguida é feita uma busca pelas informações do outro usuário que está participando da conversa e por fim é realizada uma busca pela ultima mensagem da conversa.

```

6  async get (req, res) { // retorna as informacoes do chat atraves da url
7    const { userId } = req.params
8    try {
9      const chats = await Chat.find({
10        members: { $in: [userId] }
11      }, { chatId: 1, members: 1, createdAt: 1 })
12
13      const users_id = []
14      const messages_id = []
15
16      chats.forEach(chat => {
17        users_id.push(chat.members.find(id => id === userId))
18        messages_id.push(chat._id)
19      })
20
21      const users = await User.find({ _id: { $in: users_id } },
22        { username: 1, name: 1, avatar: 1 })
23      const messages = []
24
25      for (const message_id of messages_id) {
26        const message = await Message.findOne({ chatId: { $in: message_id } },
27          { text: 1, createdAt: 1 }).sort({ createdAt: -1 })
28        messages.push(message)
29      }
30
31      const data = chats.map((chat, index) => { // Colocar no formato ideal para o cliente
32        let date = chat.createdAt
33        let lastMessage = 'Diga "oi"'
34
35        if (messages[index]) {
36          date = messages[index].createdAt
37          lastMessage = messages[index].text
38        }
39
40        return {
41          _id: chat._id,
42          members: chat.members,
43          createdAt: date,
44          lastMessage: lastMessage,
45          user: {
46            username: users[index].username,
47            name: users[index].name,
48            avatar: users[index].avatar
49          }
50        }
51      })
52
53      res.status(200).json(data)
54    } catch (error) {
55      console.log(error)
56      return res.status(500).send({ error })
57    }
58  },

```

Dados recebidos:



Retorno:

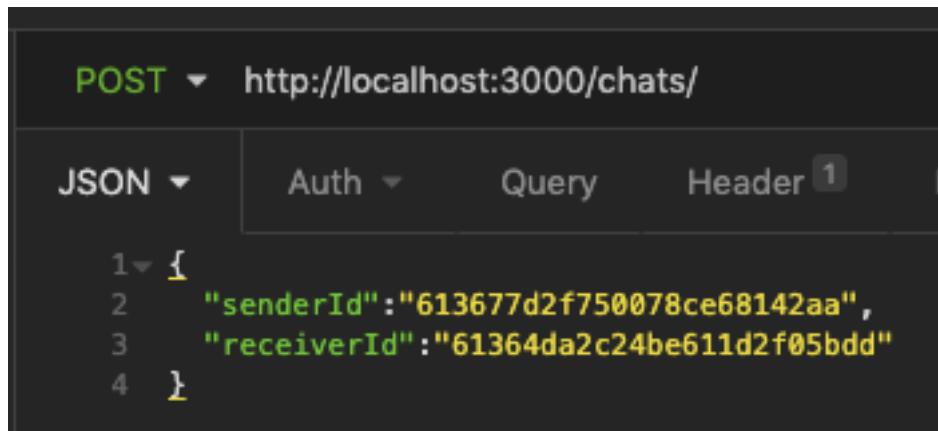
```
1 [  
2 {  
3   "_id": "615cbc96b79afb89ec92892d",  
4   "members": [  
5     "615cbc66b79afb89ec928925",  
6     "61364da2c24be611d2f05b2f"  
7   ],  
8   "createdAt": "2021-10-05T21:01:31.858Z",  
9   "lastMessage": "Jfkdk",  
10  "user": {  
11    "username": "sahudy",  
12    "name": "sahudy",  
13    "avatar": ""  
14  }  
15}  
16]
```

### 5.2.3.2 NewChat

NewChat é uma função que criar uma nova conversa, em que recebe como entrada os IDs de dois usuários e partir disso é criado um novo registro no banco de dados.

```
53 | async newChat (req, res) { // criar um novo chat atraves do id do remetente e destinatario  
54 |   const { senderId, receiverId } = req.body  
55 |   const chat = new Chat({  
56 |     members: [senderId, receiverId]  
57 |   })  
58 |  
59 >   try { ...  
60 | } catch (error) {  
61 |   return res.status(500).send({ error })  
62 | }  
63 | }  
64 |  
65 | ,|
```

Dados recebidos:

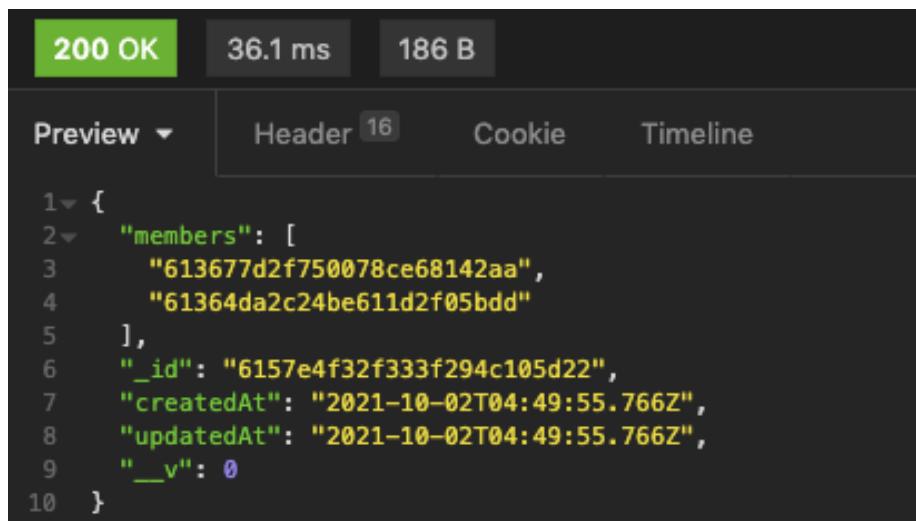


POST ▼ http://localhost:3000/chats/

JSON ▼ Auth ▼ Query Header 1

```
1 {  
2   "senderId": "613677d2f750078ce68142aa",  
3   "receiverId": "61364da2c24be611d2f05bdd"  
4 }
```

Retorno:



200 OK 36.1 ms 186 B

Preview ▼ Header 16 Cookie Timeline

```
1 {  
2   "members": [  
3     "613677d2f750078ce68142aa",  
4     "61364da2c24be611d2f05bdd"  
5   ],  
6   "_id": "6157e4f32f333f294c105d22",  
7   "createdAt": "2021-10-02T04:49:55.766Z",  
8   "updatedAt": "2021-10-02T04:49:55.766Z",  
9   "__v": 0  
10 }
```

Registro no Banco de dados:



```
> _id: ObjectId("615c648b02db9744d7afe75f")  
  ↴ members: Array  
    ↴ 0: "613677d2f750078ce68142aa"  
    ↴ 1: "61364da2c24be611d2f05bdd"  
  ↴ createdAt: 2021-10-05T14:43:23.387+00:00  
  ↴ updatedAt: 2021-10-05T14:43:23.387+00:00  
  ↴ __v: 0
```

#### 5.2.4 Controller das mensagens

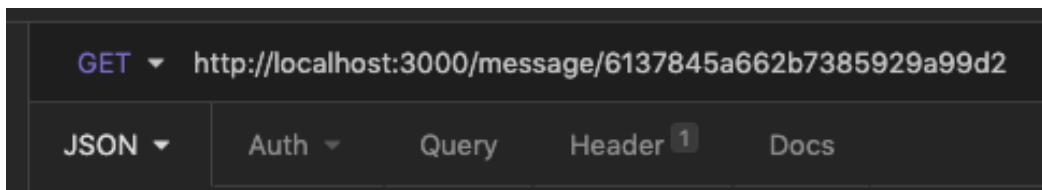
O controller das mensagens possui duas funções o Get e o newMessage.

#### 5.2.4.1 Get

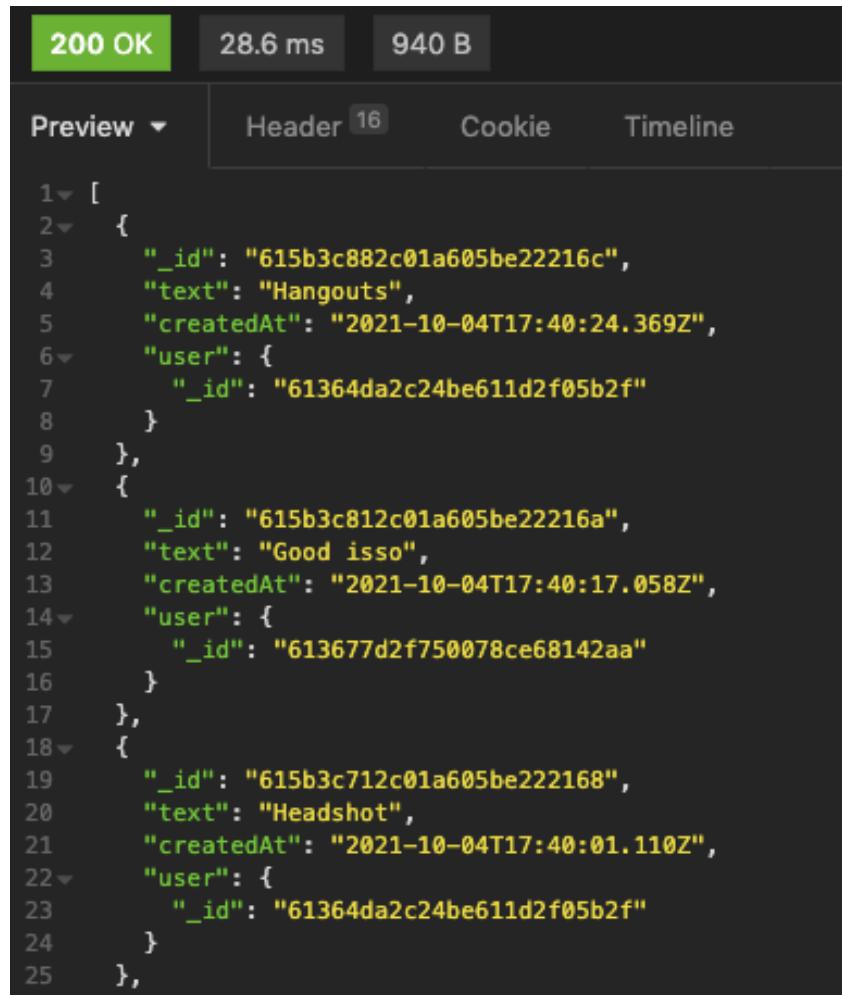
Função responsável por retorna todas as mensagens de um chat de acordo com id que recebe de entrada. É feito uma busca no banco de dados pelo id do chat na collection message, em seguida é retornado o resultado da busca.

```
4  async get (req, res) { // retorna o conteudo da mensagem atraves do id
5    const { chatId } = req.params
6    try {
7      const messages = await Message.find({ chatId }).sort({ createdAt: -1 })
8
9      const data = messages.map((message) => {
10        return {
11          _id: message._id,
12          text: message.text,
13          createdAt: message.createdAt,
14          user: {
15            _id: message.sender
16          }
17        }
18      })
19
20      res.status(200).json(data)
21    } catch (error) {
22      console.log(error)
23      if (error) { return res.status(500).send({ error: error }) }
24    }
25  },
```

Dados recebidos:



Retorno:



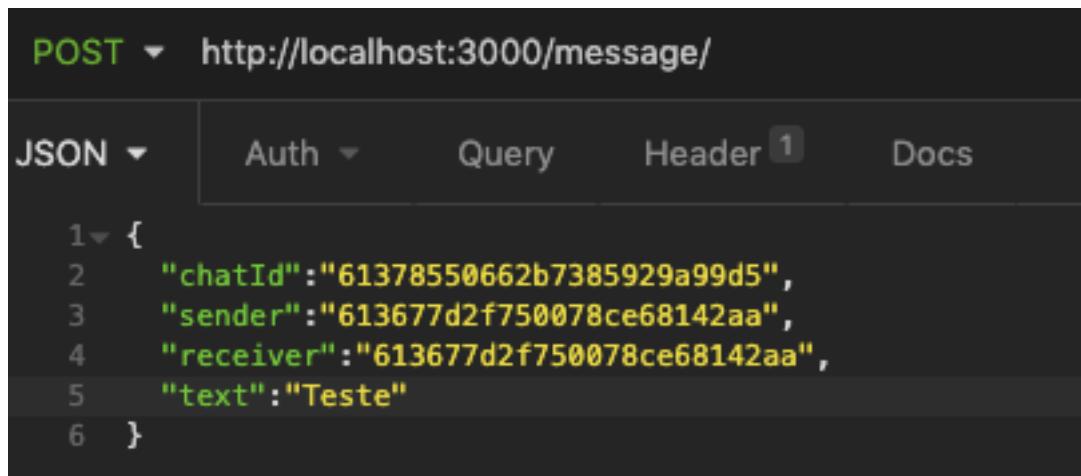
The screenshot shows a Postman API response. At the top, it displays a green button labeled "200 OK", a grey box showing "28.6 ms", and another grey box showing "940 B". Below this, there are tabs: "Preview" (which is selected and has a dropdown arrow), "Header" (with a count of 16), "Cookie", and "Timeline". The main content area contains a JSON array with three elements, each representing a message. The messages are as follows:

```
1 [  
2 {  
3   "_id": "615b3c882c01a605be22216c",  
4   "text": "Hangouts",  
5   "createdAt": "2021-10-04T17:40:24.369Z",  
6   "user": {  
7     "_id": "61364da2c24be611d2f05b2f"  
8   },  
9 },  
10 {  
11   "_id": "615b3c812c01a605be22216a",  
12   "text": "Good isso",  
13   "createdAt": "2021-10-04T17:40:17.058Z",  
14   "user": {  
15     "_id": "613677d2f750078ce68142aa"  
16   },  
17 },  
18 {  
19   "_id": "615b3c712c01a605be222168",  
20   "text": "Headshot",  
21   "createdAt": "2021-10-04T17:40:01.110Z",  
22   "user": {  
23     "_id": "61364da2c24be611d2f05b2f"  
24   },  
25 }]
```

#### 5.2.4.2 NewMessage

Tem a funcionalidade de criar uma nova mensagem, recebe de entrada o id do chat, remetente, destinatário e por fim o texto da mensagem. A partir desses dados de entrada é criado um novo registro no banco de dados.

Dados recebidos:

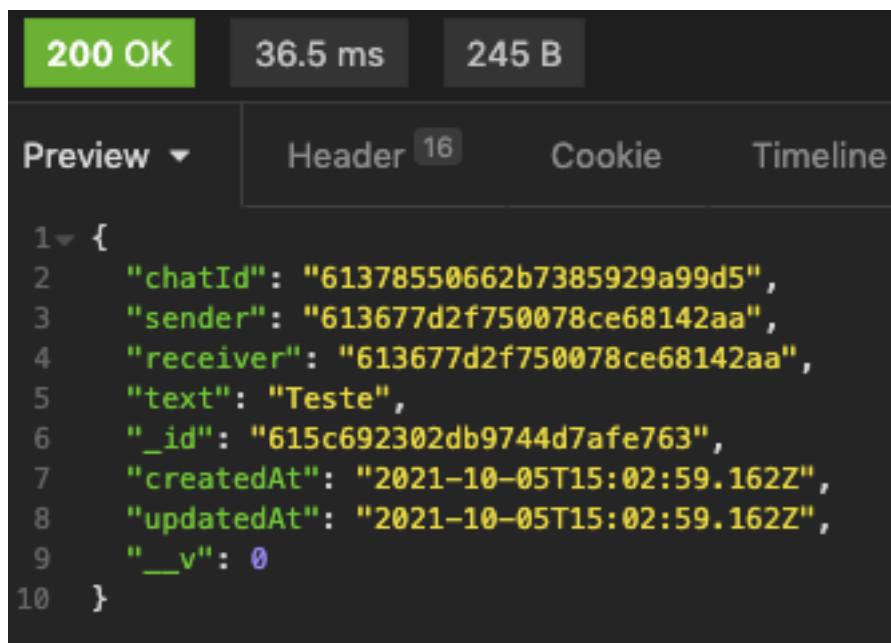


POST ▾ http://localhost:3000/message/

JSON ▾ Auth ▾ Query Header 1 Docs

```
1▼ {  
2   "chatId": "61378550662b7385929a99d5",  
3   "sender": "613677d2f750078ce68142aa",  
4   "receiver": "613677d2f750078ce68142aa",  
5   "text": "Teste"  
6 }
```

Retorno:



200 OK 36.5 ms 245 B

Preview ▾ Header 16 Cookie Timeline

```
1▼ {  
2   "chatId": "61378550662b7385929a99d5",  
3   "sender": "613677d2f750078ce68142aa",  
4   "receiver": "613677d2f750078ce68142aa",  
5   "text": "Teste",  
6   "_id": "615c692302db9744d7afe763",  
7   "createdAt": "2021-10-05T15:02:59.162Z",  
8   "updatedAt": "2021-10-05T15:02:59.162Z",  
9   "__v": 0  
10 }
```

Registro no Banco de dados:



```
_id: ObjectId("615c692302db9744d7afe763")  
chatId: "61378550662b7385929a99d5"  
sender: "613677d2f750078ce68142aa"  
receiver: "613677d2f750078ce68142aa"  
text: "Teste"  
createdAt: 2021-10-05T15:02:59.162+00:00  
updatedAt: 2021-10-05T15:02:59.162+00:00  
__v: 0
```

## 6. IMAGENS DAS COLLECTIONS CRIADAS

### Message:

The screenshot shows the MongoDB Compass interface with the 'myFirstDatabase.messages' collection selected. The collection has 8 documents and a total size of 1.56KB. The results show three documents with their IDs and some fields highlighted in red.

```
_id: ObjectId("615cbc96b79af89ec928930")
chatId: "615cbc96b79af89ec92892d"
sender: "615cbc66b79af89ec928925"
receiver: "61364da2c24be611d2f05b2f"
text: "01"
createdAt: 2021-10-05T20:59:14.637+00:00
updatedAt: 2021-10-05T20:59:14.637+00:00
__v: 0

_id: ObjectId("615cbd08b79af89ec92893a")
chatId: "615cbc96b79af89ec92892d"
sender: "615cbc66b79af89ec928925"
receiver: "61364da2c24be611d2f05b2f"
text: "diff"
createdAt: 2021-10-05T21:00:56.451+00:00
updatedAt: 2021-10-05T21:00:56.451+00:00
__v: 0

_id: ObjectId("615bd12b79af89ec92893c")
chatId: "615cbc96b79af89ec92892d"
sender: "61364da2c24be611d2f05b2f"
receiver: "615cbc66b79af89ec928925"
text: "dkasdo"
createdAt: 2021-10-05T21:01:06.507+00:00
```

### Chat:

The screenshot shows the MongoDB Compass interface with the 'myFirstDatabase.chats' collection selected. The collection has 1 document and a total size of 147B. The result shows one document with its ID and some fields highlighted in red.

```
_id: ObjectId("615cbc96b79af89ec92892d")
members: Array
createdAt: 2021-10-05T20:59:02.710+00:00
updatedAt: 2021-10-05T20:59:02.710+00:00
__v: 0
```

## User:

The screenshot shows the MongoDB Compass interface. On the left, there's a sidebar with 'DATABASES: 1' and 'COLLECTIONS: 3'. The 'myFirstDatabase' database is selected, showing three collections: 'chats', 'messages', and 'users'. The 'users' collection is currently selected. At the top right, there are buttons for 'Default landing', 'VISUALIZE YOUR DATA', and 'REFRESH'. Below the collection name, it says 'COLLECTION SIZE: 1.45KB' and 'TOTAL DOCUMENTS: 7' with 'INDEXES TOTAL SIZE: 72KB'. There are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'FILTER' button with a query '{ field: 'value' }' is present. On the right, there are buttons for 'INSERT DOCUMENT', 'OPTIONS', 'Apply', and 'Reset'. The 'QUERY RESULTS 1-1 OF 1' section displays a single document with the following fields and values:

```
_id: ObjectId("61364da2c24be611d2f05b2f")
username: "luispeixoto"
name: "Luis"
password: "$2b$08$R6MisWw/S0bhfGKVfMx609iECX0l.i9zig5lqy.osn05r92i2tR0"
avatar: ""
> followers: Array
> followings: Array
__v: 0
```

## 7. TESTE DE REQUISIÇÕES

### 7.1 Métricas

Para realização dos testes de requisições utilizaremos o artillery.io para a criação de usuários virtuais e os disparos de requisições. O artillery.io se trata de uma ferramenta para realização de teste de desempenho, onde realizaremos o teste de carga no banco de dados com o objetivo de analisar a quantidade de requisições que o banco de dados aguenta.

Analisaremos as seguintes:

- Latência de Leitura/Escrita
- Latência média Leitura/Escrita
- Mediana da latência de Leitura/Escrita
- Desvio padrão da latência de Leitura/escrita
- Número de Falhas e Sucessos

### 7.2 Ambientes testados

Os testes foram realizados pelos três integrantes do grupo em ambientes distintos:

<b>Integrante</b>	<b>Luis Fernando Peixoto</b>	<b>Ralf Mateus</b>	<b>Yuri Garcia</b>
<b>Sistema Operacional</b>	macOS Big Sur	Windows 10	Windows 10
<b>Processador</b>	i7-9700K 3.60GHz	i5-4210U 1.70GHz	i7-7700K 4.20GHz
<b>Memória-ram</b>	32GB	12,0 GB DDR3L	16,0 GB
<b>Velocidade de Internet</b>	100MB	70MB	200MB
<b>Tráfego</b>	Leve	Pesado	Pesado

### 7.3 Cenários

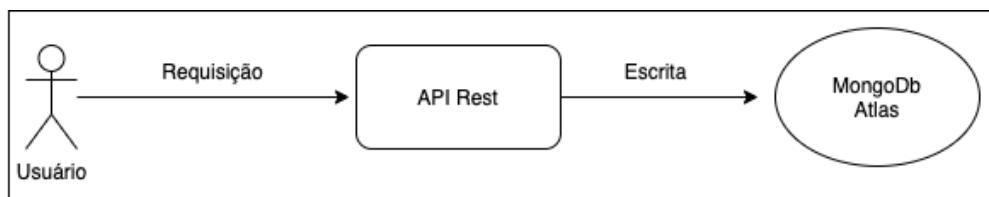
Os testes serão divididos em envios de requisições síncronas (requisições enviadas uma de cada vez conforme são respondidas pelo servidor) e assíncronas (Envio de diversas requisições em paralelo).

#### 7.3.1 Requisições assíncronas

Serão realizados testes em dois cenários assíncronos, o primeiro cenário será realizado com um único usuário realizando requisições de escrita. O segundo teste será feito com dois usuários de escritas, enviando requisições simultâneas.

##### 7.3.1.1 Um usuário realiza o envio de requisições de escrita

- Durante 1 segundo envia 250, 500 e 1000 requisições.
- Durante 1 minuto envia 1000(16/s), 5000(83/s) e 10000(166/s) requisições.
- Durante 5 minutos envia 10.000(33/s), 25.000(83/s) e 50.000(166/s) requisições.



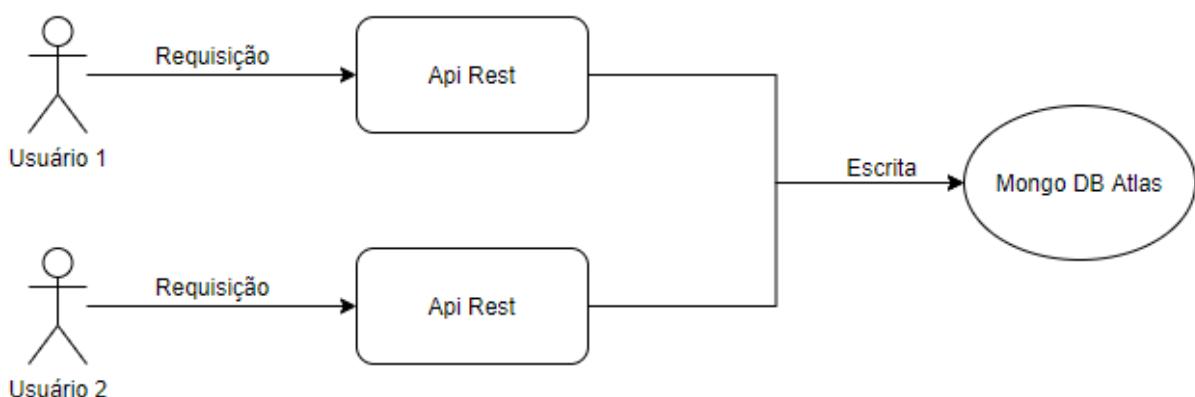
### 7.3.1.2 Dois usuários realizam o envio de requisições de escrita simultâneas

- **Usuário 1 - Escrita**

- Durante 1 segundo serão feitas 125, 250 e 500 requisições.
- Durante 1 minuto serão feitas 500(8/s), 2.500(42/s) e 5.000(83/s) requisições.
- Durante 5 minutos serão feitas 5.000(16/s), 12.500(42/s) e 25.000(83/s)

- **Usuário 2 - Escrita**

- Durante 1 segundo serão feitas 125, 250 e 500 requisições.
- Durante 1 minuto serão feitas 500(8/s), 2.500(42/s) e 5.000(83/s) requisições.
- Durante 5 minutos serão feitas 5.000(16/s), 12.500(42/s) e 25.000(83/s)

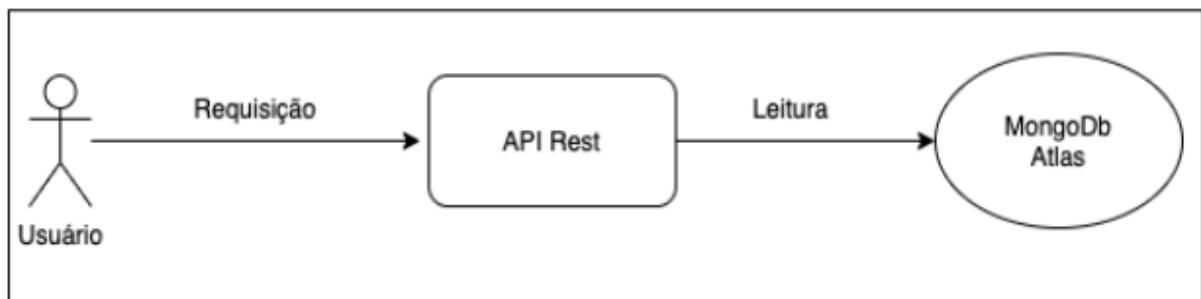


### 7.3.2 Requisições síncronas

O teste síncrono contou com três cenários, o primeiro e o segundo cenário foram realizados por um usuário enviando requisições de leitura no primeiro cenário e de escrita no segundo cenário. O terceiro cenário ocorre com a distribuição de carga em operações de 50% de leitura e 50% de escritas, onde o Usuário 1 envia requisições de leitura, enquanto o Usuário 2 envia requisições de escritas.

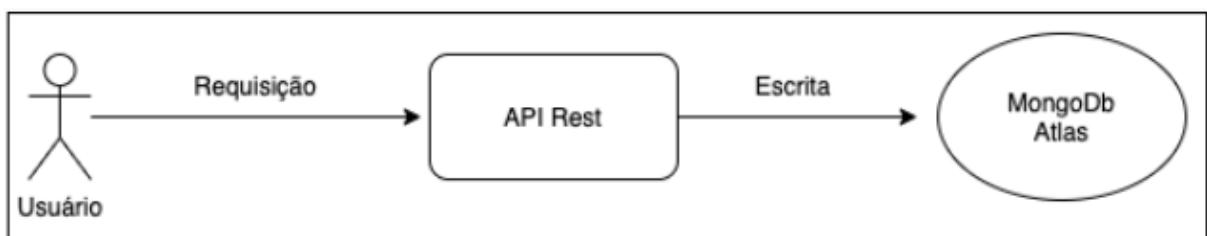
### 7.3.2.1 Um usuário realiza o envio de requisições de leitura

- 1.000, 10.000 e 25.000 requisições.



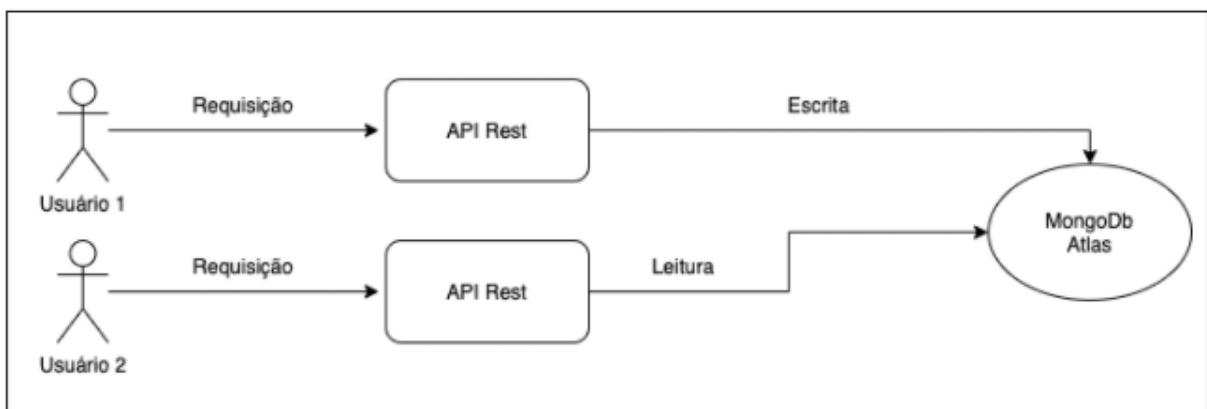
### 7.3.2.2 Um usuário realiza o envio de requisições de escrita

- 1.000, 10.000 e 25.000 requisições.



### 7.3.2.3 Dois usuários realizam o envio de requisições de leitura e escrita

- **Usuário 1 - Leitura**
  - As requisições de leitura serão enviadas juntamente com as requisições do usuário 2
- **Usuário 2 - Escrita**
  - 1.000, 10.000 e 25.000 requisições.



## 7.4 Realização dos testes

Para realizar os testes foi usada a ferramenta artillery.io em que permite realizar os disparos de requisições. Para usar o artillery.io foi criado um script de configuração de testes onde é colocado a rota da aplicação a ser disparada requisições, o tempo de duração do teste e quantos disparos por segundo vão ser feitos naquele tempo.

```
test.yml
1 config:
2   target: "<ROTA PARA O BANCO DE DADOS>"
3   phases:
4     - duration: <DURAÇÃO DO TESTE EM SEGUNDOS>
5       arrivalCount: <NÚMERO DE DESPAROS POR SEGUNDO>
6   scenarios:
7     - flow:
8       - get:
9         url: "/"
```

Junto a essa ferramenta foram utilizadas flags dentro do código para que fosse extraído o tempo de cada execução e assim pudéssemos calcular uma média e gráficos do tempo de execução.

Os dados gerados pelo código fonte, foram armazenados em arquivos de textos como o da imagem abaixo em que mostra o tempo de cada requisição de escrita:

```
testSendMessage.txt
1 0.044
2 0.024
3 0.022
4 0.023
5 0.023
6 0.023
7 0.022
8 0.023
9 0.023
10 0.023
11 0.026
12 0.023
13 0.028
14 0.038
15 0.026
16 0.023
17 0.027
18 0.025
```

#### 7.4.1 Teste de leitura

```
const time1 = new Date().getUTCMilliseconds()
const messages = await Message.find({ chatId }).sort({ createdAt: -1 })
const time2 = new Date().getUTCMilliseconds()
const result = time2 - time1
```

Na imagem acima podemos ver um exemplo. Essa imagem é uma parte do código de leitura de documentos e nele foi adicionado as variáveis time1 e time2 que nos resultam o tempo que levou para ser retornados os documentos de cada requisição feita pelo artillery.io.

#### 7.4.2 Teste de escrita

```
const data = { // objeto que tem
  chatId: chatId, // o id do chat
  sender: sender, // o id do remetente
  receiver: receiver, // o id do destinatário
  text: lorem.generateParagraphs(1) // texto gerado
}
const newMessage = new Message(data)

const time1 = Date.now()
await newMessage.save()
const time2 = Date.now()

const result = time2 - time1 // duração do tempo para salvar
```

As mensagens enviadas nas requisições de escrita são geradas automaticamente com conteúdo e tamanho aleatório.

## 7.5 Resultados

Os testes foram submetidos aos cenários do tópico anterior e foram realizados pelos três integrantes do grupo do projeto na qual possuem conexões com a internet em velocidades diferentes, para que possamos verificar a interferência da latência nos testes.

### 7.5.1 Requisições assíncronas

#### 7.5.1.1 Um usuário realiza o envio de requisições de escrita

Integrante	Disparos	Tempo (s)	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Taxa de falha	Falhas	Sucessos
Luis	250	1s	1,14	1,21	0,5	64%	160	90
Ralf	250	1s	2,91	2,83	0,45	98,8%	247	3
Yuri	250	1s	1,57	1,4	0,54	70%	175	75
Luis	500	1s	1,97	2,12	1,18	72,2%	361	139
Ralf	500	1s	1,52	1,66	0,75	89,9%	449	51
Yuri	500	1s	1,3	1,75	1,17	67%	335	165
Luis	1.000	1s	5,22	4,85	2,75	84%	840	160
Ralf	1.000	1s	2,37	2,32	0,96	92,1%	921	79
Yuri	1.000	1s	2,28	2,24	1,56	80,4%	804	196
Luis	1.000(16/s)	60s	0,03	0,03	0,01	0%	0	1.000
Ralf	1.000(16/s)	60s	0,03	0,03	0,02	0%	0	1.000
Yuri	1.000(16/s)	60s	0,03	0,03	0,01	0%	0	1.000
Luis	5.000(83/s)	60s	0,02	0,03	0,03	0%	0	5.000
Ralf	5.000(83/s)	60s	0,28	0,33	0,25	0%	0	5.000
Yuri	5.000(83/s)	60s	0,03	0,03	0,003	0%	0	5.000
Luis	10.000(166/s)	60s	19,33	19,31	11,36	0%	0	10.000
Ralf	10.000(166/s)	60s	22,39	22,44	14,47	0%	0	10.000
Yuri	10.000(166/s)	60s	20,26	20,16	11,58	0%	0	10.000
Luis	10.000(33/s)	300s	0,04	0,04	0,034	0%	0	10.000

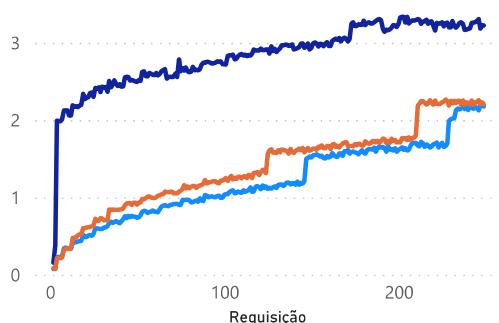
Ralf	10.000(33/s)	300s	0,04	0,07	0,162	0%	0	10.000
Yuri	10.000(33/s)	300s	0,04	0,04	0,011	0%	0	10.000
Luis	25.000(83/s)	300s	0,04	0,04	0,011	0%	0	25.000
Ralf	25.000(83/s)	300s	0,07	0,071	1,64	0%	0	25.000
Yuri	25.000(83/s)	300s	0,04	0,04	0,009	0%	0	25.000
Luis	50.000(166/s)	300s	101,32	35,09	18,789	0%	0	50.000
Ralf	50.000(166/s)	300s	38,35	101,39	57,16	0%	0	50.000
Yuri	50.000(166/s)	300s	55,99	52,98	24,972	0%	0	50.000

Como podemos ver nas tabelas quanto maior o tempo que temos para enviar menos falhas ocorrem, vale dizer que as falhas ocorrem quando uma requisição ultrapassa o intervalo de tempo determinado no teste. Isso acontece porque o MongoDB Atlas tem um limite de 100 processos por segundo e se o número de processos ultrapassar esse limite (o que é o caso desse teste) ele cria uma fila de espera que será executada só no próximo segundo, isso fica mais fácil de ver no gráfico, e isso causa um número maior de falhas quando temos apenas um segundo para mandar mais de 100 requisições, já que o Atlas irá criar um gargalo e todos as outras requisições depois das 100 primeiras falharam no teste de tempo. Em intervalo de tempo maior mesmo com o gargalo e com um maior número de requisições ainda sim a taxa de falhas chega a 0%.

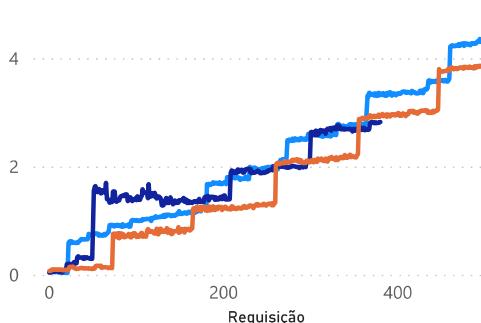
Luis Fernando Peixoto    Ralf Cruz    Yuri Garcia

### 1 segundo

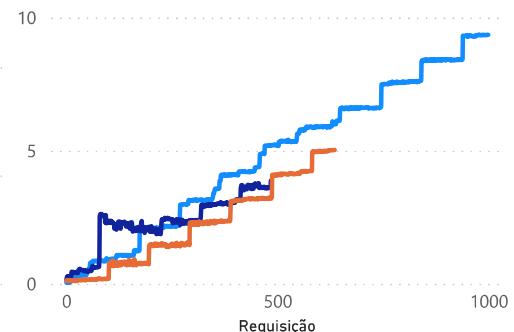
250 Requisições por segundo



500 Requisições por segundo

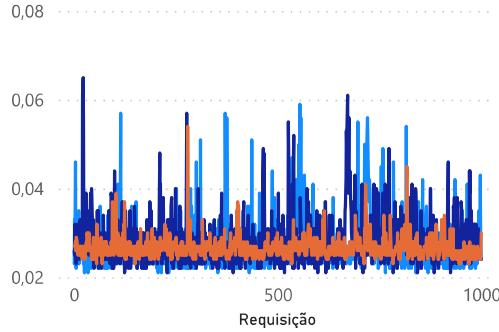


1.000 Requisições por segundo

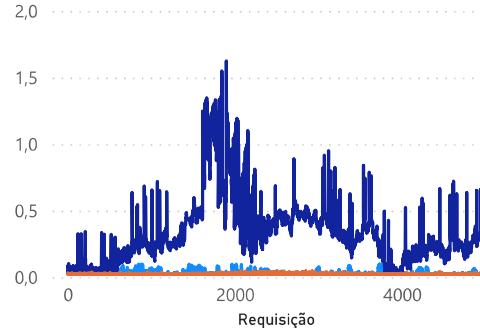


### 1 Minuto

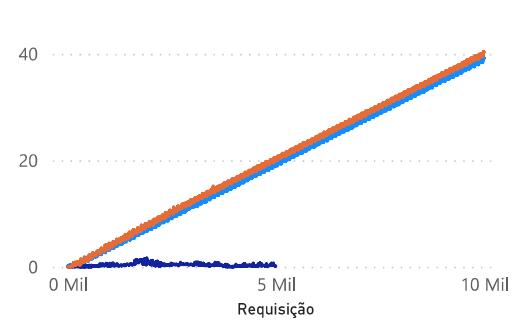
1.000 Requisições (16/s)



5.000 Requisições (83/s)

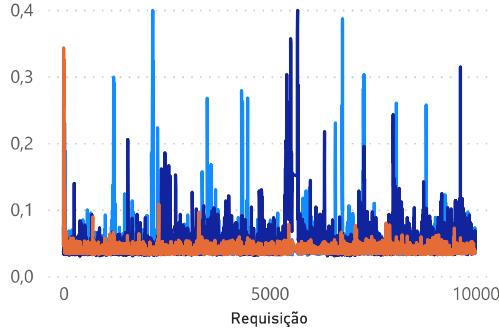


10.000 Requisições (166/s)

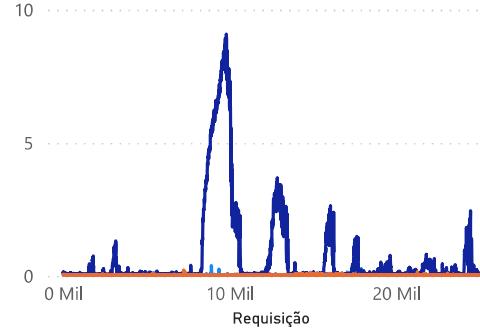


### 5 Minutos

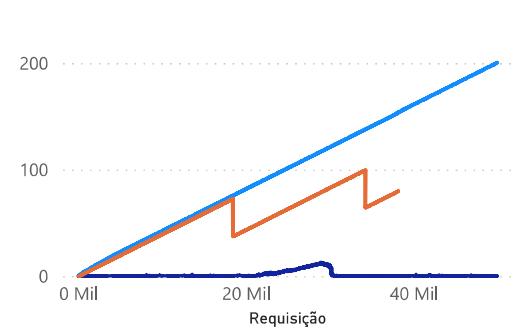
10.000 Requisições (33/s)



25.000 Requisições (83/s)



50.000 Requisições (166/s)



A partir dos gráficos podemos ver as filas que o MongoDB Atlas forma, que são essas linhas em formato de cascata, que se formam aproximadamente a cada 100 requisições, assim como está dito na documentação do Atlas. E em relação as linhas da cor azul forte podemos ver que como Ralf estava realizando os testes em um computador e uma velocidade de internet inferior, suponhamos que o tempo que o computador demora para fazer as requisições compensa o limite de processos do Atlas e isso faz com que não fique tão explícito o tempo de espera das filas.

#### 7.5.1.2 Dois usuários realizam o envio de requisições de escrita e leitura

Integrante	Disparos	Tempo (s)	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Taxa de falha	Falhas	Sucessos
<b>Usuário 1</b>								
Luis	125	1s	0,03	0,21	0,255	0%	0	125
Ralf	125	1s	0,07	0,08	0,041	88%	110	15
Yuri	125	1s	0,06	0,11	0,098	0%	0	125
Luis	250	1s	0,59	0,55	0,454	19,2%	48	202
Ralf	250	1s	0,06	0,07	0,025	0%	0	250
Yuri	250	1s	0,61	0,77	0,696	31,20%	78	172
Luis	500	1s	0,94	1,77	1,457	43%	285	215
Ralf	500	1s	0,07	0,12	0,148	29%	143	357
Yuri	500	1s	3,22	3,96	2,235	92,20%	461	39
Luis	500(8/s)	60s	0,03	0,03	0,01	0%	0	1000
Ralf	500(8/s)	60s	0,03	0,03	0,02	0%	0	1000
Yuri	500(8/s)	60s	0,03	0,03	0,01	0%	0	1000
Luis	2.500(42/s)	60s	0,92	1,05	0,749	0%	0	2.500
Ralf	2.500(42/s)	60s	0,06	0,07	0,046	0%	0	2.500
Yuri	2.500(42/s)	60s	0,04	0,05	0,058	0%	0	2.500
Luis	5.000(83/s)	60s	18,91	19	11,407	0%	0	5000
Ralf	5.000(83/s)	60s	20,06	20,08	12,048	0%	0	5000
Yuri	5.000(83/s)	60s	17,73	17,87	11,419	0%	0	5000

Luis	5.000(16/s)	300s	0,04	0,04	0,021	0%	0	5000
Ralf	5.000(16/s)	300s	0,04	0,09	0,206	0%	0	5000
Yuri	5.000(16/s)	300s	0,04	0,04	0,006	0%	0	5000
Luis	12.500(42/s)	300s	0,04	0,04	0,035	0%	0	12500
Ralf	12.500(42/s)	300s	0,06	0,26	0,484	0%	0	12500
Yuri	12.500(42/s)	300s	0,04	0,04	0,04	0%	0	12500
Luis	25.000(83/s)	300s	99,62	99,52	57,344	0%	0	25000
Ralf	25.000(83/s)	300s	101,21	101,6	60,978	0%	0	25000
Yuri	25.000(83/s)	300s	55,74	52,86	25,217	0%	0	25000

### Usuário 2

Luis	125	1s	0,49	0,51	0,355	0%	0	125
Ralf	125	1s	0,07	0,08	0,036	100%	125	0
Yuri	125	1s	0,27	0,34	0,266	0%	0	125
Luis	250	1s	1,34	1,35	0,497	72,40%	181	69
Ralf	250	1s	0,06	0,07	0,026	0%	0	250
Yuri	250	1s	2,14	2,04	0,728	88,80%	222	28
Luis	500	1s	5,3	4,76	1,846	97%	485	15
Ralf	500	1s	0,08	0,14	0,18	70%	348	152
Yuri	500	1s	1,3	1,84	1,804	27%	137	363
Luis	500(8/s)	60s	0,03	0,04	0,042	0%	0	500
Ralf	500(8/s)	60s	0,05	0,06	0,021	0%	0	500
Yuri	500(8/s)	60s	0,04	0,04	0,007	0%	0	500
Luis	2.500(42/s)	60s	0,03	0,1	0,184	0%	0	2500
Ralf	2.500(42/s)	60s	0,06	0,07	0,043	0%	0	2500
Yuri	2.500(42/s)	60s	0,04	0,05	0,068	0,2%	5	2495
Luis	5.000(83/s)	60s	19,32	19,4	11,414	0%	0	5000
Ralf	5.000(83/s)	60s	0,06	0,07	0,046	0%	0	5000
Yuri	5.000(83/s)	60s	19,29	19,36	11,459	0%	0	5000

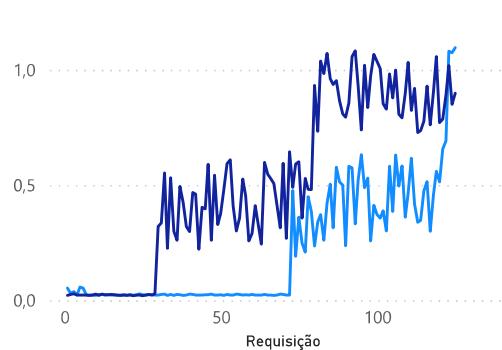
Luis	5.000(16/s)	300s	0,04	0,04	0,021	0%	0	5000
Ralf	5.000(16/s)	300s	0,04	0,09	0,21	0%	0	5000
Yuri	5.000(16/s)	300s	0,04	0,04	0,006	0%	0	5000
Luis	12.500(42/s)	300s	0,04	0,04	0,034	0%	0	12500
Ralf	12.500(42/s)	300s	0,06	0,26	0,472	0%	0	12500
Yuri	12.500(42/s)	300s	0,04	0,04	0,028	0%	0	12500
Luis	25.000(83/s)	300s	100,37	100,28	57,333	0%	0	25000
Ralf	25.000(83/s)	300s	108,18	109,57	61,951	0%	0	25000
Yuri	25.000(83/s)	300s	56,62	53,56	25,098	0%	0	25000



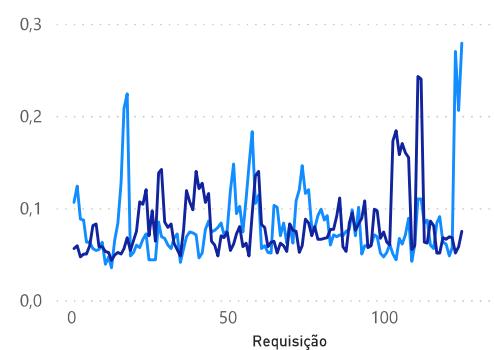
1 Segundo

250 - Requisições por segundo

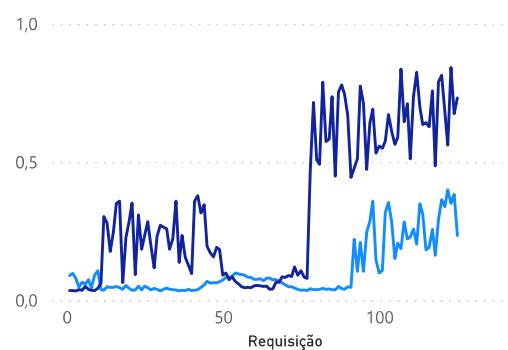
Luis Fernando Peixoto



Ralf Cruz

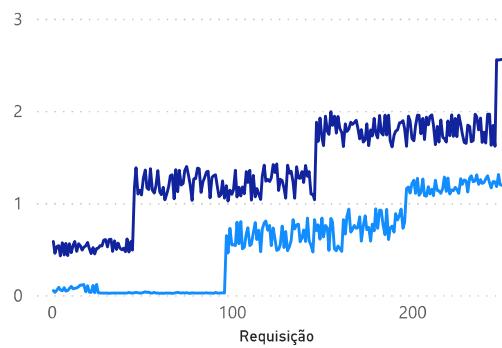


Yuri Garcia

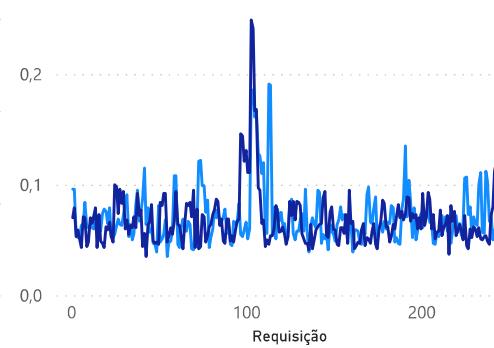


500 Requisições por segundo

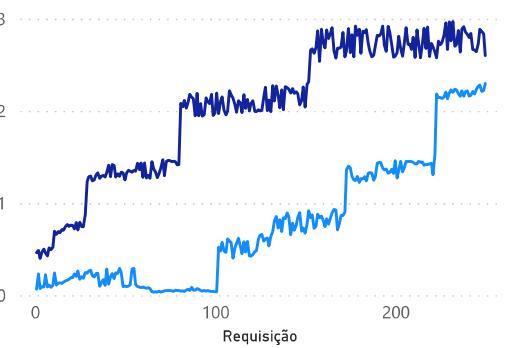
Luis Fernando Peixoto



Ralf Cruz

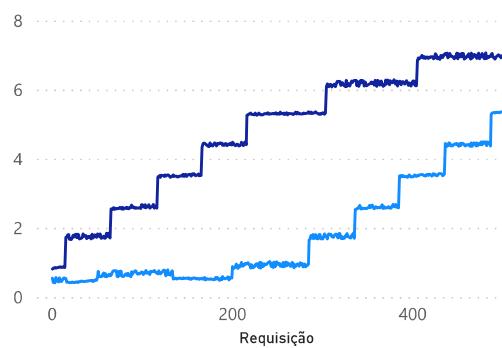


Yuri Garcia

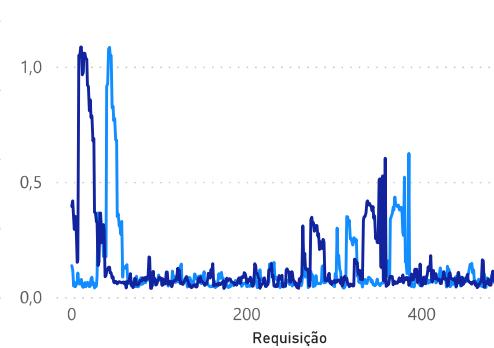


1.000 Requisições por segundo

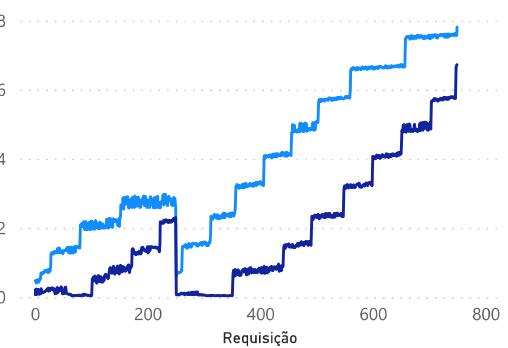
Luis Fernando Peixoto



Ralf Cruz



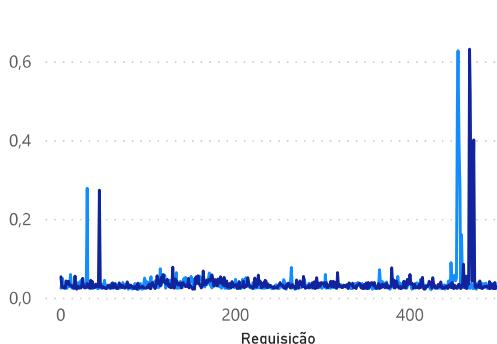
Yuri Garcia



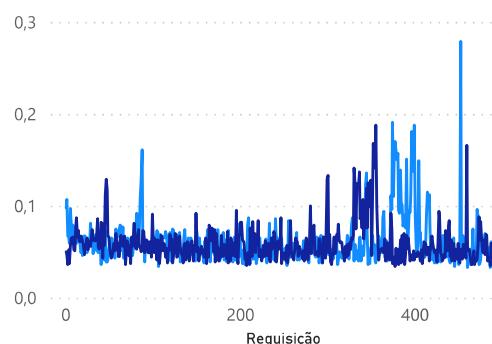
1 Minuto

1.000 Requisições (16/s)

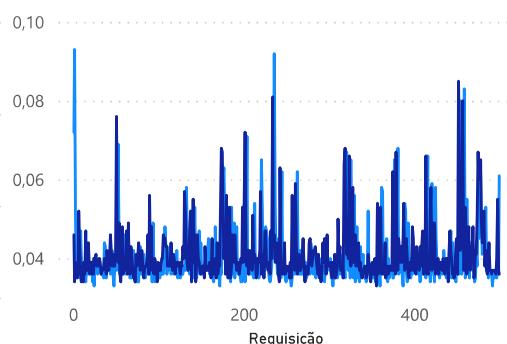
Luis Fernando Peixoto



Ralf Cruz

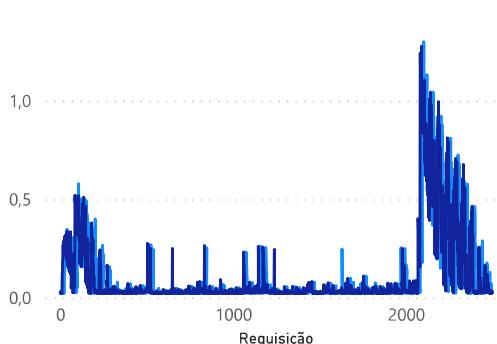


Yuri Garcia

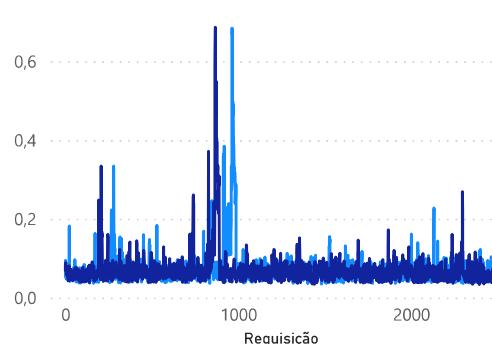


5.000 Requisições (83/s)

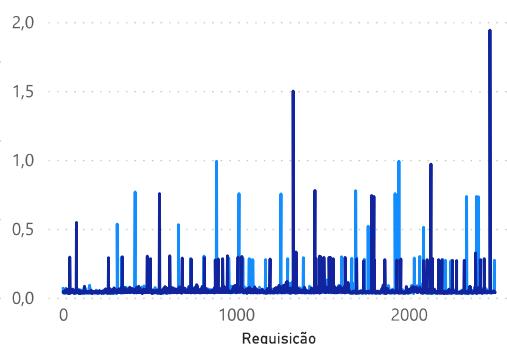
Luis Fernando Peixoto



Ralf Cruz

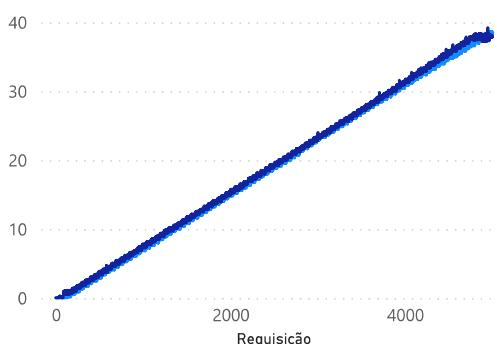


Yuri Garcia

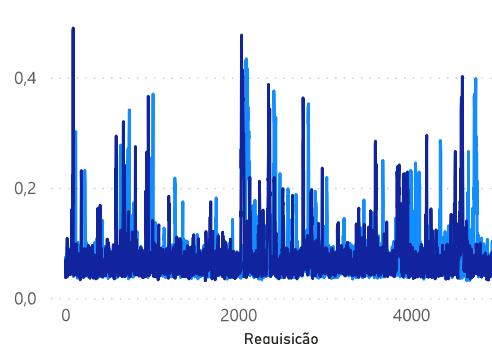


10.000 Requisições (166/s)

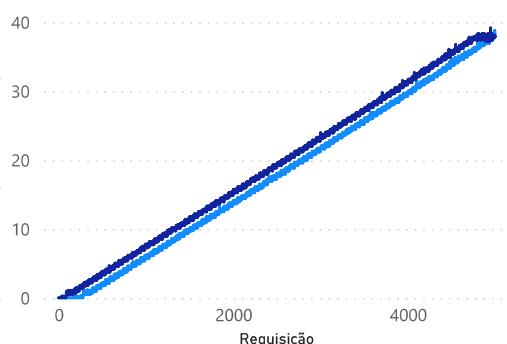
Luis Fernando Peixoto



Ralf Cruz



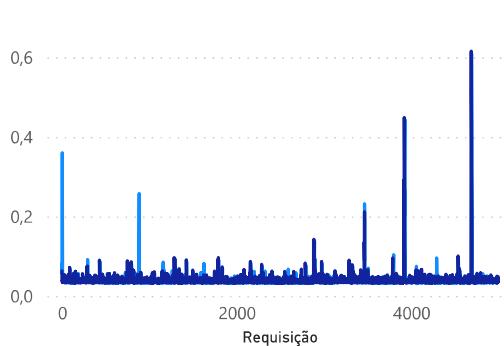
Yuri Garcia



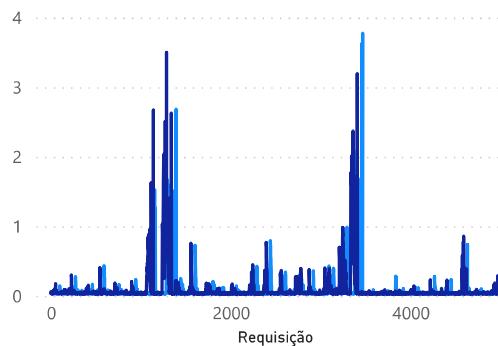
## 5 Minutos

10.000 Requisições (33/s)

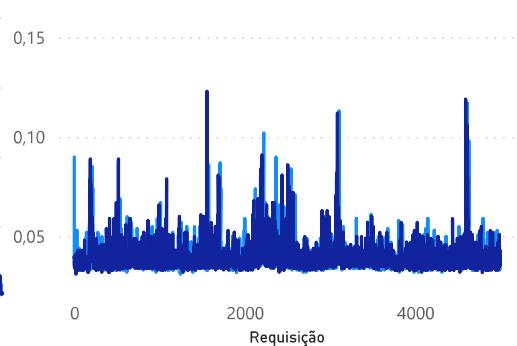
Luis Fernando Peixoto



Ralf Cruz

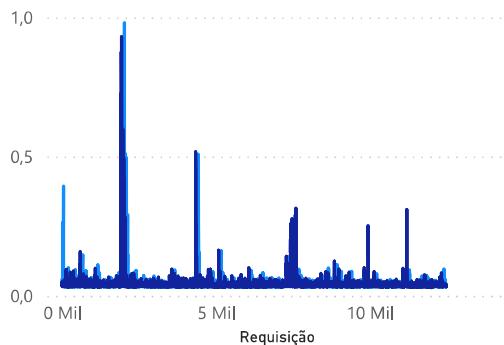


Yuri Garcia

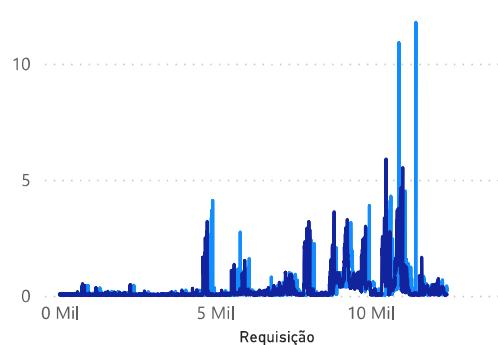


25.000 Requisições (83/s)

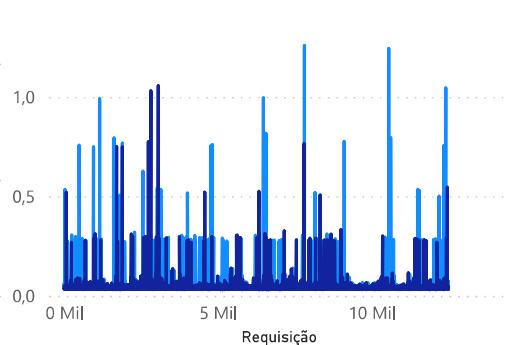
Luis Fernando Peixoto



Ralf Cruz

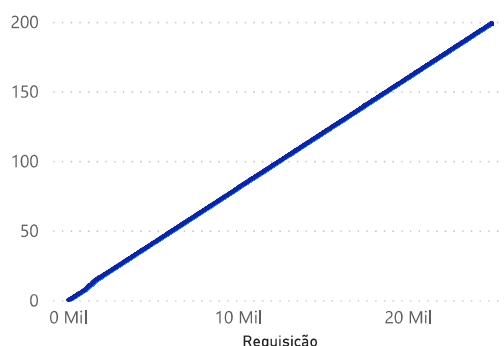


Yuri Garcia

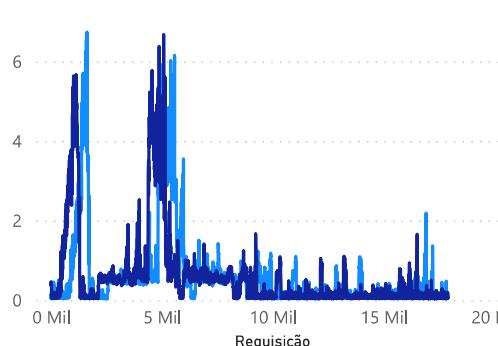


50.000 Requisições (166/s)

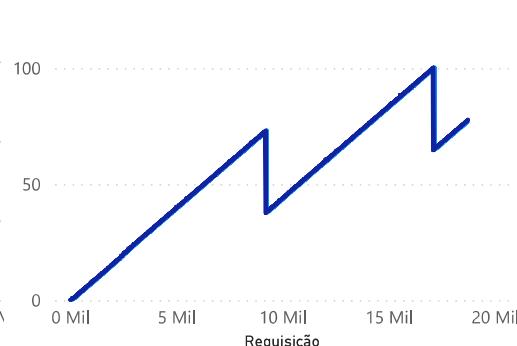
Luis Fernando Peixoto



Ralf Cruz



Yuri Garcia



Como pode ser visto tanto na tabela quanto nos gráficos o que foi dito antes sobre a fila e os gargalos ainda se aplica ao teste com dois usuários de escrita, mas o que podemos ver é que quando existem dois usuários, o que entra na fila primeiro vem a ter um tempo menor do que o outro, mas o gráfico dos dois usuários é bem parecido só que o usuário que entra depois vai demorar um pouco mais. Já o nosso teste feito em um computador mais inferior faz com que os usuários cheguem até a fila em um tempo mais parecido e fazendo com que eles alternam os lugares na fila do MongoDB Atlas e assim os dois usuários tem um tempo de processamento que fica bem mais parecido e chegam a convergir e isso é o que faz os gráficos de Ralf ficarem com as linhas mais “embaralhadas” e é o que deveria acontecer em todos os testes, mas como a ferramenta acaba iniciando um usuário primeiro que o outro, mesmo apenas alguns milésimos de segundos, as linhas dos outros gráficos acabam ficando um pouco afastadas e não “embaralhadas”.

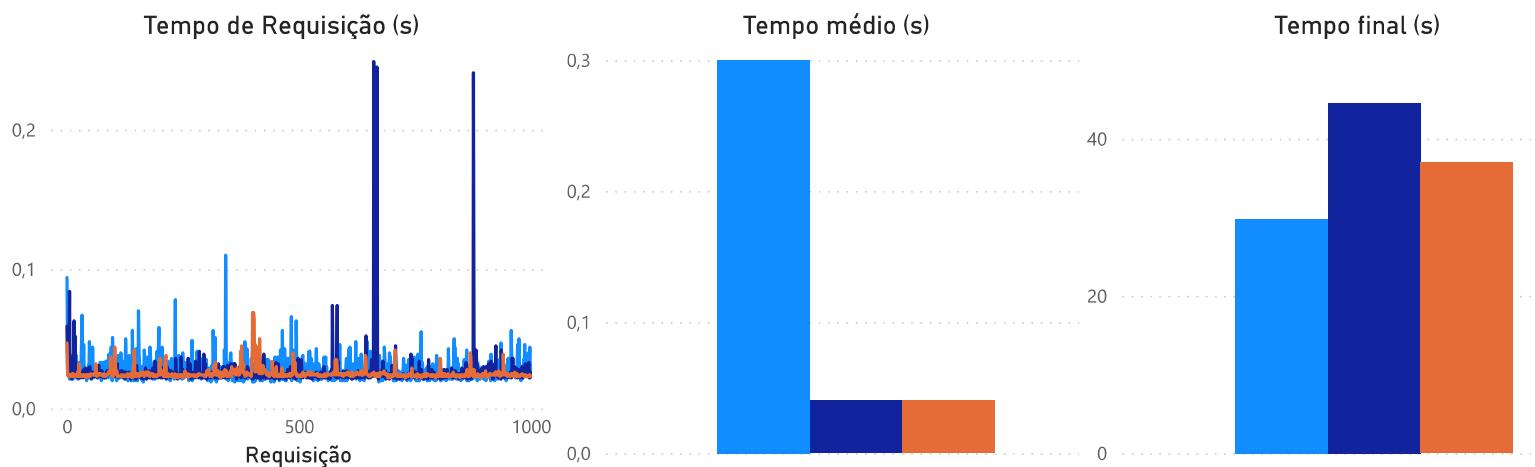
### 7.5.2 Requisições síncronas

#### 7.5.2.1 Um usuário realiza o envio de requisições de escrita

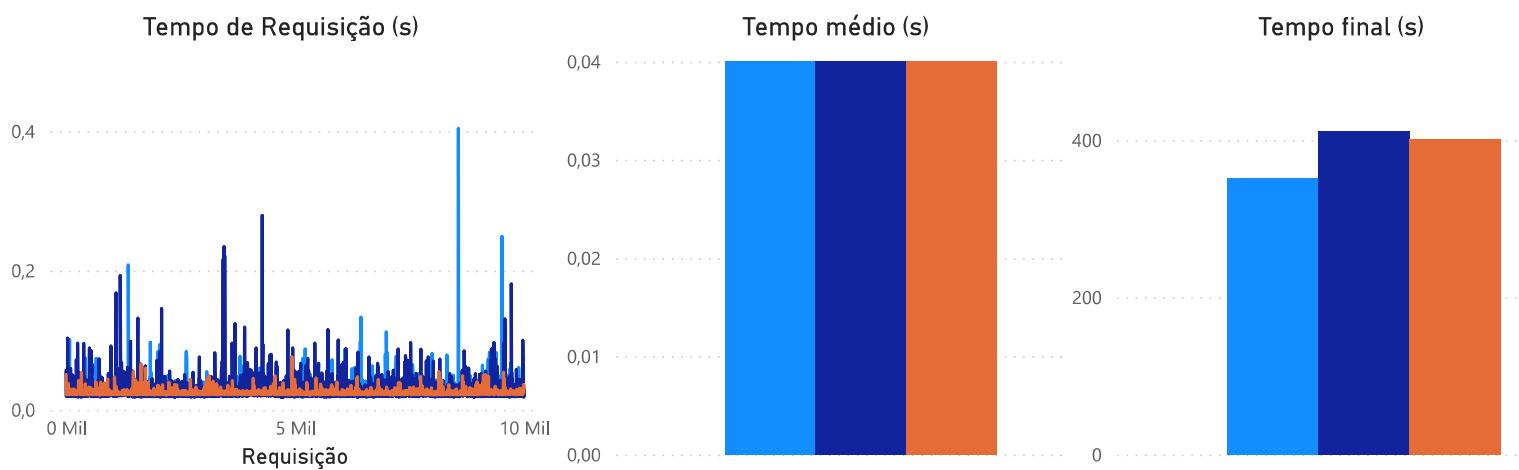
Integrante	Disparos	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Tempo Final (s)
Luis	1.000	0,02	0,03	0,013	25,5
Ralf	1.000	0,03	0,03	0,008	27,66
Yuri	1.000	0,02	0,02	0,004	24,9
Luis	10.000	0,02	0,03	0,008	252,32
Ralf	10.000	0,03	0,03	0,011	271,44
Yuri	10.000	0,02	0,02	0,003	243,65
Luis	25.000	0,02	0,03	0,01	630,59
Ralf	25.000	0,02	0,02	0,009	531,96
Yuri	25.000	0,02	0,03	0,021	626,45

Nos testes síncronos os gargalos não acontecem e isso faz com que tenhamos uma média de tempo bem baixa e podemos também ter uma ideia do tempo total que leva certas quantidades de requisições para serem cumpridas se elas não acontecessem de forma paralela.

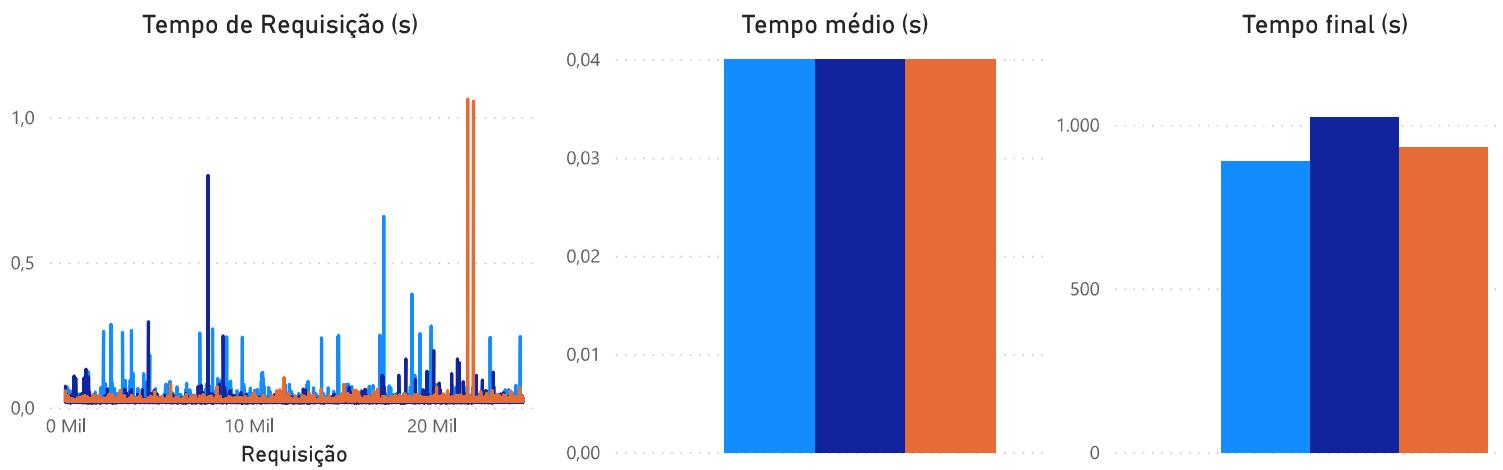
## 1.000 Requisições



## 10.000 Requisições



## 25.000 Requisições

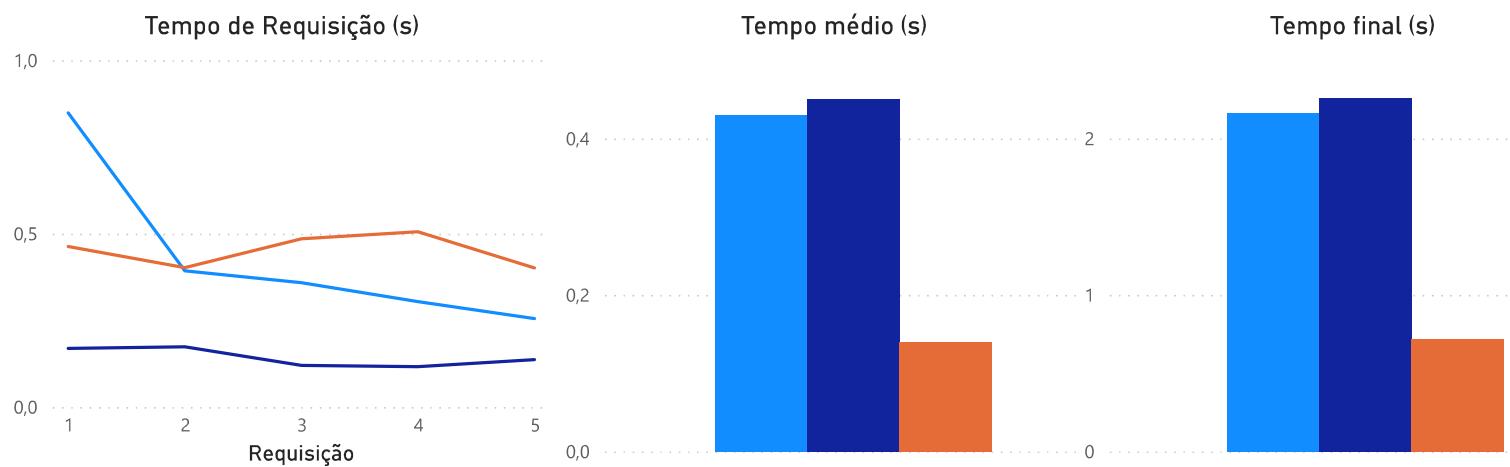


### 7.5.2.2 Um usuário realiza o envio de requisições de leitura

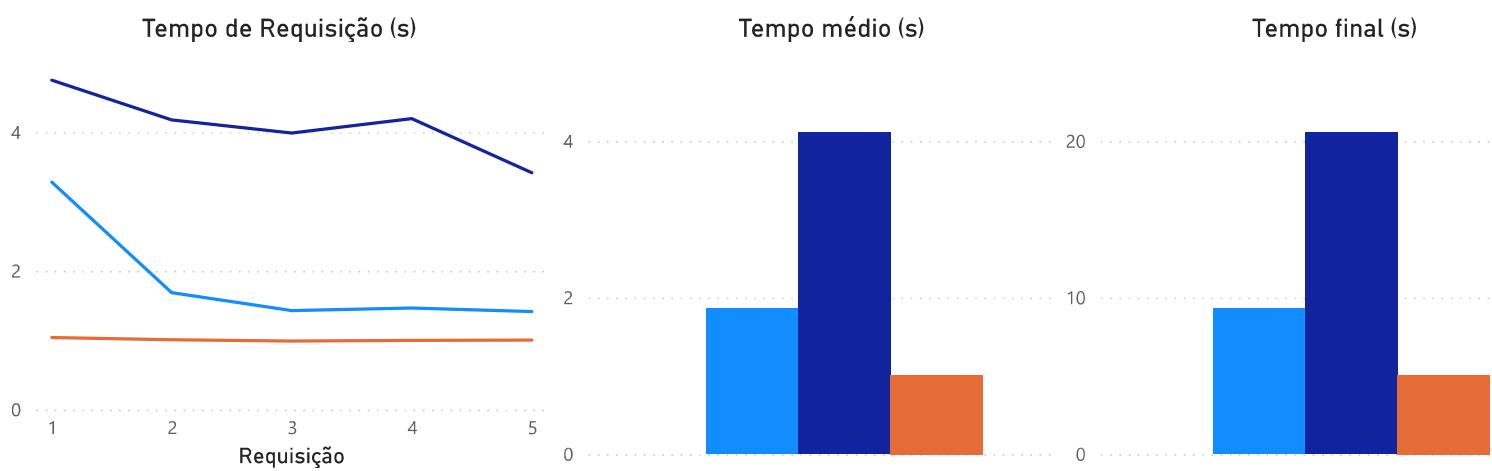
Integrante	Qtd. de documentos	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Tempo Final (s)
Luis	1.000	0,36	0,43	0,239	2,16
Ralf	1.000	0,46	0,45	0,048	2,26
Yuri	1.000	0,14	0,14	0,027	0,72
Luis	10.000	1,47	1,86	0,805	9,28
Ralf	10.000	4,18	4,11	0,48	20,53
Yuri	10.000	1	1,01	0,019	5,04
Luis	25.000	4,59	4,84	0,210	22,88
Ralf	25.000	8,86	8,84	0,088	44,2
Yuri	25.000	2,45	2,53	0,138	12,65

Luis Fernando Peixoto    Ralf Cruz    Yuri Garcia

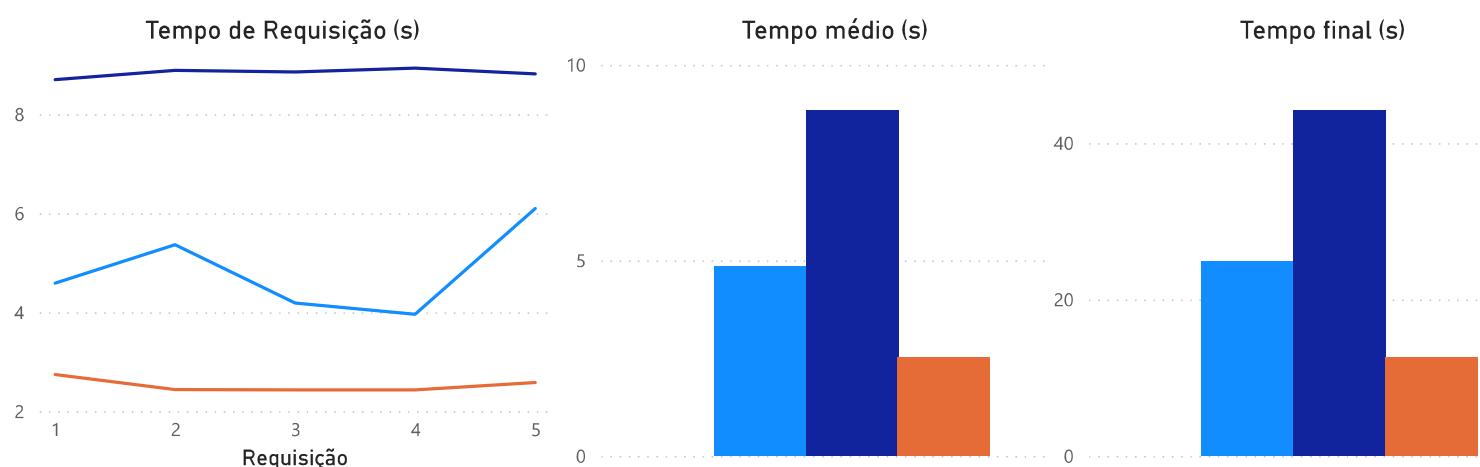
## 1.000 Documentos



## 10.000 Documentos



## 25.000 Documentos



Observando a tabela de leitura e os gráficos podemos concluir que à medida que aumenta o número de documentos lidos consequentemente o tempo de leitura aumenta. Também é possível observar que como Yuri possui uma velocidade com a internet maior que os demais integrantes do projeto, obteve-se tempos de leituras menores, portanto teve melhores resultados, mostrando que a velocidade com a internet interfere no tempo de leitura.

#### 7.5.2.3 Dois usuários realizam o envio de requisições de leitura e escrita

Usuário de Leitura						
Integrante	Qtd. de documentos	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Tempo Final (s)	Ponto de Falha
Luis	1.000	0,17	0,17	0,11	64,97	-
Ralf	1.000	1,12	1,87	1,747	54,16	-
Yuri	1.000	1,22	1,91	1,803	45,78	-
Luis	10.000	0,32	0,52	0,51	368,77	-
Ralf	10.000	1,69	2,82	2,892	87,27	4826
Yuri	10.000	2,04	3,1	2,286	102,25	2430
Luis	25.000	0,43	1,18	6,482	985,78	18615
Ralf	25.000	1,52	2,73	2,788	90,11	1503
Yuri	25.000	2,32	3,35	2,897	70,35	2545
Usuário de Escrita						
Integrante	Disparos	Mediana (s)	Tempo Médio (s)	Desvio Padrão (s)	Tempo Final (s)	
Luis	1.000	0,3	0,3	0,1	29,75	
Ralf	1.000	0,04	0,04	0,025	44,53	
Yuri	1.000	0,04	0,04	0,004	37	
Luis	10.000	0,3	0,4	0,2	351,27	
Ralf	10.000	0,04	0,04	0,058	412,11	
Yuri	10.000	0,04	0,04	0,055	401,79	
Luis	25.000	0,03	0,04	0,03	889,19	
Ralf	25.000	0,04	0,04	0,018	1024,88	
Yuri	25.000	0,04	0,04	0,024	931,95	



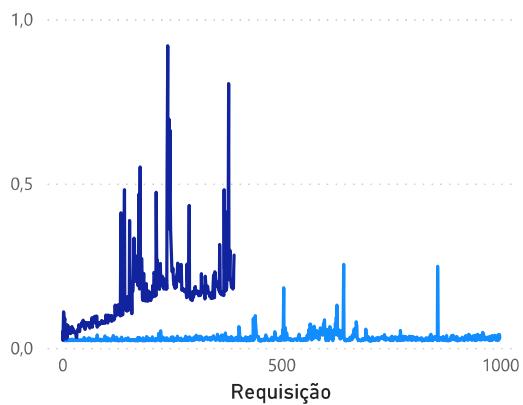
Usuário de Escrita



Usuário de Leitura

## 1.000 Requisições

Luis Fernando Peixoto



Ralf Cruz

6

4

2

0

500

1000

Ralf Cruz

Requisição

Yuri Garcia

6

4

2

0

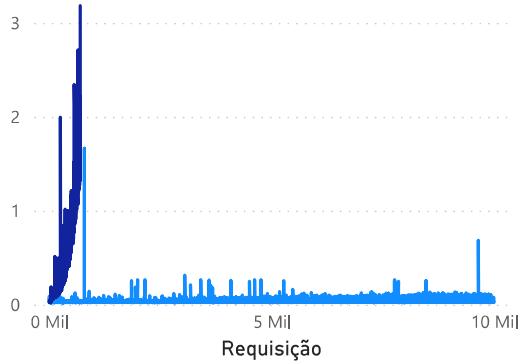
500

1000

Requisição

## 10.000 Requisições

Luis Fernando Peixoto



Ralf Cruz

10

5

0

0 Mil

5 Mil

10 Mil

Requisição

Yuri Garcia

10

5

0

0 Mil

5 Mil

10 Mil

10000

Requisição

## 25.000 Requisições

Luis Fernando Peixoto



Ralf Cruz

100

50

0

0 Mil

10 Mil

20 Mil

Requisição

Yuri Garcia

10

5

0

0 Mil

10 Mil

20 Mil

Requisição

Como visto a leitura demora mais do que a escrita e por isso quando tentamos fazer as requisições de leitura e escrita simultâneas a leitura acaba falhando nos testes em algum momento, já que a cada nova requisição de leitura temos muito mais documentos para ler do que na requisição de leitura anterior, e com isso acaba gerando um gargalo na qual o MongoDB Atlas não consegue suportar as requisições de leitura e ocorre a falha.

## 8. CONCLUSÃO

Durante os testes realizados o MongoDB Atlas se mostrou confiável, uma vez que não foram perdidos nenhuma requisição de escrita, mesmo que o tempo da requisição tenha ultrapassado o intervalo definido, ocasionando uma falha nos testes, porém todas as requisições de escrita foram completadas e consequentemente armazenadas no banco de dados.

Percebemos também ao realizar os testes, principalmente no cenário com dois usuários realizando requisições de escrita e um outro usuário realizando requisições de leitura, que em um determinado momento do teste as requisições de leitura falhavam enquanto as requisições de escrita continuavam sendo realizadas normalmente, mesmo que tentássemos retomar os testes de leitura o MongoDB Atlas não permitia e ocorria a falha novamente. Não conseguimos nem mesmo acessar o painel de controle do cluster que estava sendo testado, só era possível acessar novamente quando todas as requisições de escrita fossem concluídas. Suponhamos que isso tenha ocorrido devido ao MongoDB Atlas priorizar as requisições de escrita para que nenhum dado sejam perdidos.

Com base nos dados obtidos através dos testes, pode ser concluído que o MongoDB Atlas versão gratuito é um bom banco de dados para ser usados em projetos pequenos. Como vimos ele é de graça e apesar da versão gratuita do Atlas ter um limite relativamente pequeno de requisições por segundo, ele consegue ter um bom desempenho mesmo com muitas requisições. Mas ele não é indicado para grandes empresas ou projetos maiores, que envolvam muitas pessoas enviando

muitas requisições simultâneas, pois acaba tendo um baixo desempenho e podendo ocorrer algumas falhas de leitura quando possui muitos documentos para serem lidos.

## 9. CRONOGRAMA DO PROJETO

TAREFA	Data de inicio	Data de término
Desenvolvimento da interface	03/09/2021	03/09/2021
Desenvolvimento da API	04/09/2021	07/09/2021
Integração com socket.io	08/09/2021	18/09/2021
Integração da API e o socket.io com a interface	19/09/2021	05/10/2021
Realização dos testes de requisições	06/10/2021	17/11/2021

## 10. REPOSITÓRIO DA APLICAÇÃO NO GITHUB

Aplicativo: <https://github.com/LuisPeixoto/campion>

API-REST: <https://github.com/LuisPeixoto/campion-server>

## 11. BIBLIOGRAFIA

LOAD testing - Socket.io. [S. I.], 2021. Disponível em: <https://socket.io/docs/v4/load-testing/>. Acesso em: 26 set. 2021.

TESTING Socket.IO - Artillery.io. [S. I.], 2021. Disponível em:  
<https://artillery.io/docs/guides/guides/socketio-reference.html>. Acesso em: 26 set. 2021.

ATLAS M0 (Free Cluster), M2, and M5 Limitations. [S. I.], 2021. Disponível em:  
<https://docs.atlas.mongodb.com/reference/free-shared-limitations/#operational-limitations>. Acesso em: 9 nov. 2021.

WHY Artillery?. [S. I.], 2021. Disponível em:  
<https://artillery.io/docs/guides/overview/why-artillery.html>. Acesso em: 26 set. 2021.