**MongoDB com mongoose no node**

1. Iremos fazer um sistema para agendamento em um calendário.
2. O núcleo do sistema será composta pela biblioteca Calendar.

**INICIANDO O PROJETO E INSTALANDO AS DEPENDÊNCIAS**

1. npm init -y
2. npm install express –save
3. npm install ejs –save
4. npm install body-parser –save
5. npm install mongoose –save
6. npm install nodemon --save

**SETUP PADRÃO NO INDEX.JS**

1. Na pasta do projeto crie as pastas view e public.
2. Na pasta do projeto crie o arquivo index.js
3. Faça as configurações abaixo
4. //importando o express
5. *const* express = require('express');
6. //importando o body-parser para formulário e json
7. *const* bodyParser = require('body-parser');
8. //importando o mongoose para utilizar o banco de dados mongodb
9. *const* mongoose = require('mongoose');
10. //criando a instancia do express em um app
11. *const* app = express();
12. //configurando o uso de arquivos estáticos para a página do calendar utilizar
13. //o css e o javascript
14. app.use(express.static('public'));
15. //configurano o app pra utilizar o express em formulários e o json
16. app.use(express.urlencoded({extended: false} ));
17. app.use(express.json());
18. //configurando o framework da view engine
19. app.set('view-engine','ejs');
20. //configurando a conexao padrao do mongodb
21. mongoose.connect("mongodb://localhost:27017/agendamento",{useNewUrlParser: true, useUnifiedTopology: true});
22. //rota principal
23. app.get('/',(*req*, *res*)*=>*{
24. *res*.send("Olá!");
25. })
26. //iniciando o servidor
27. app.listen(8080,()*=>*{
28. console.log("Servidor rodando");
29. })

**CRIANDO O BANCO DE DADOS NO MONGODB**

1. Vamos criar o banco de dados chamado agendamento
2. Abra o cmd, digite mongo
3. Digite : use agendamento.

**INSTALANDO O FULLCALLENDAR NO PROJETO**

1. Vá na página <http://fullcalendar.io>
2. Entre em Get Started
3. Vá em Pre-built Bundles
4. Clique “Initialize with Script Tags”
5. Clique dowload: “fullcalendar-5.10.1.zip”
6. Após baixar copie todo conteúdo da pasta lib para dentro da pasta public do projeto.

**CRIANDO A VIEW DE CRIAÇÃO DE CONSULTAS**

1. Na pasta view crie o arquivo create.ejs
2. Configure a estrutura básica do html:5
3. <!DOCTYPE html>
4. <html lang="en">
5. <head>
6. <meta charset="UTF-8">
7. <meta http-equiv="X-UA-Compatible" content="IE=edge">
8. <meta name="viewport" content="width=device-width, initial-scale=1.0">
9. <title>Cadastro</title>
10. </head>
11. <body>
13. </body>
14. </html>
15. Vamos colocar o bootstrap na página através do cdn, copie para o head o link do cdn do bootstrap.
16. Copie também os links do pooper e jquery e cole abaixo do body.
17. <!DOCTYPE html>
18. <html lang="en">
19. <head>
20. <meta charset="UTF-8">
21. <meta http-equiv="X-UA-Compatible" content="IE=edge">
22. <meta name="viewport" content="width=device-width, initial-scale=1.0">
23. <title>Cadastro</title>
24. <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">
26. </head>
27. <body>
29. </body>
30. <script src="https://code.jquery.com/jquery-3.6.0.slim.min.js" integrity="sha256-u7e5khyithlIdTpu22PHhENmPcRdFiHRjhAuHcs05RI=" crossorigin="anonymous"></script>
31. <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
32. <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery.maskedinput/1.4.1/jquery.maskedinput.min.js" type="text/javascript"></script>
34. </html>

**CRIANDO FORMULÁRIO DE CADASTRO**

1. <body>
2. <h1 class="content">Cadastro de consulta</h1>
3. <hr>
4. <form class="form">
5. <input type="text" name="name" id="name" placeholder="Nome do paciente" class="form-control">
6. <input type="text" name="email" id="email" placeholder="email@email.com" class="form-control">
7. <input type="text" name="cpf" id="cpf" placeholder="CPF" class="form-control">
8. <input type="text" name="description" id="description" placeholder="Descrição" class="form-control">
9. <input type="date" name="date" id="date" class="form-control" placeholder="01/01/2012">
10. <input type="time" name="time" id="time" class="form-control">
11. <button class="btn btn-success btn-block">Agendar</button>
12. </form>

**CRIANDO MÁSCARAS PARA CAMPOS DE FORMULÁRIOS**

1. Iremos utilizar um plugin jquery-mask-plugin, entre no endereço <https://igorescobar.github.io/jQuery-Mask-Plugin/docs.html>.
2. Clique em download now. E iremos baixar um zip.
3. Na pasta dist vamos copiar somente o arquivo jquery.mask.min e colar na pasta public do projeto.
4. Na view onde a mascara será utilizada devemos criar um script importando o jquery-mask.
5. Crie outra tag script e dentro dela vamos configurar as mácaras conforme modelos do endereço citado acima.
6. Crie um script também para importar as máscaras de input do jquery.
7. <!--Inicio Script para utilizar máscaras no formulário-->
9. <script scr="../public/jquery.mask.min.js"></script>
10. <script>
11. $(document).ready(*function*(*$*){
12. *$*('.cpf').mask('999.999.999-99');
13. });
14. </script>
15. <!--Fim Script para utilizar máscaras no formulário    , {reverse: true}); -->

**CRIANDO O MODEL DE CONSULTA MÉDICA**

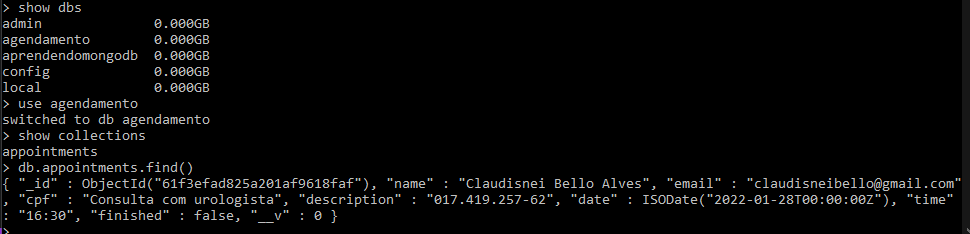
1. Crie a pasta models e um arquivo chamado Appointment.js.
2. Neste arquivo importe o moongose e crie o sckema para o model.
3. *const* mongoose = require('mongoose');
4. *const* appointment = new mongoose.Schema({
5. name: *String*,
6. email: *String*,
7. cpf: *String*,
8. description: *String*,
9. date: *Date*,
10. time: *String*,
11. finished: *Boolean*
12. });
13. module.exports = appointment;

**CRIANDO O SERVIÇE**

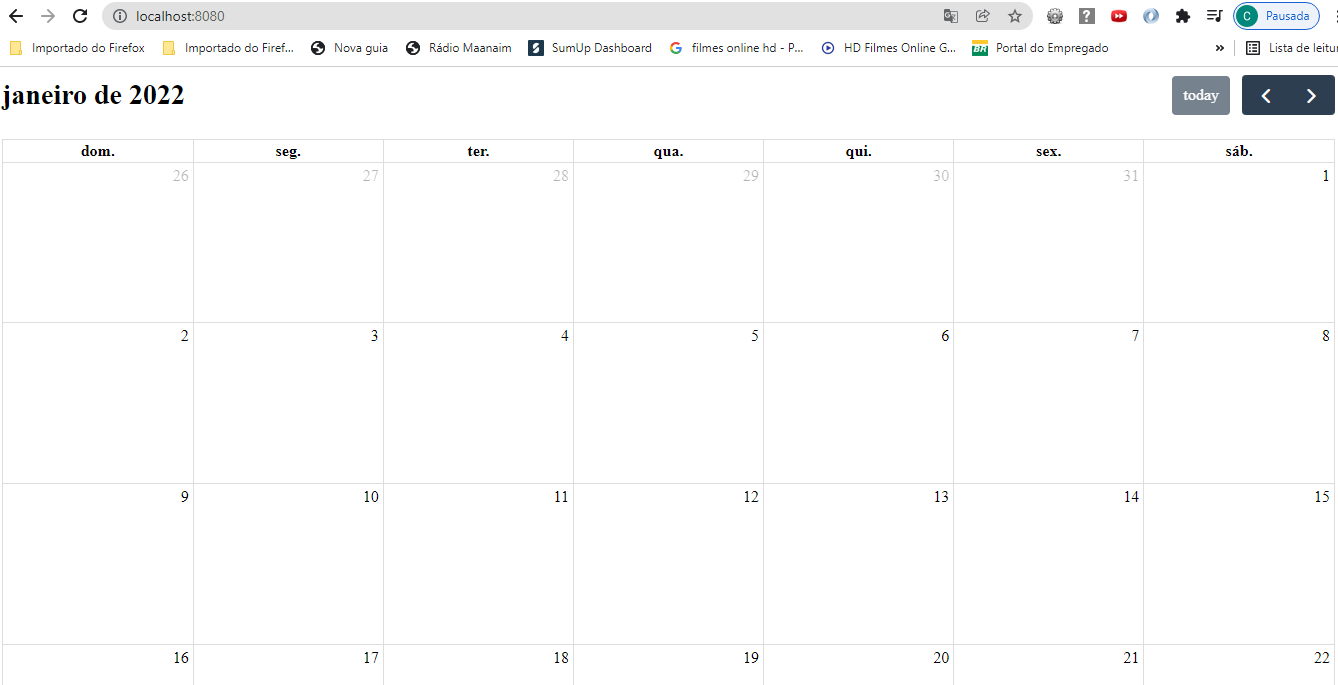
1. Iremos utilizar a arquitetura de services.
2. Vamos criar a pasta services e um arquivo chamado AppointmentService.js
3. Esse novo arquivo será uma classe com métodos.
4. Crie uma variável importando o schema Appointment.
5. Crie a variável mongoose que irá receber o ‘mongoose’.
6. *var* appointment = require('../models/Appointment');
7. *var* mongoose = require('mongoose');
8. crie o model AppModel, onde passamos o nome da tabela que será criada e o schema
9. *const* AppoModel = mongoose.model("Appointment",appointment);
10. crie a classe AppointmentService com um método assíncrono ‘create’ que irá receber todos os campos do formulário.
11. Dentro deste método crie uma variável newAppo que irá receber o model criado AppModel. Passamos todos os valores dos inputs como parâmetros para o AppModel.
12. Chame o método save do ‘model’ newApp.
13. Como o método é assíncrono, vamos chamar este método com o await.

**METODO CREATE**

1. *class* AppointmentService{
2. async create(*name*, *email*, *description*, *cpf*, *date*, *time*){
3. *var* newAppo = new AppoModel({
4. name: *name*;
5. email: *email*,
6. description: *description*,
7. cpf: *cpf*,
8. date: *date*,
9. time: *time*,
10. finished: false
11. });
12. try {
13. await newAppo.save();
14. return true;
15. } catch (error) {
16. console.log(error);
17. return false;
18. }
19. }
20. }
21. *module*.*exports* = new AppointmentService();
22. c
23. vamos importar para o index.js o service que criamos.
24. No arquivo index.js digite
25. //importando o service de consulta medica
26. *const* appointmentService = require('./services/AppointmentService');
27. Vamos criar uma rota para que quando o usuário clicar no botão do formulário os dados sejam salvos no banco de dados.
28. //rota para pegar os dados do formulário e realizar o cadastro no
29. //banco de dados
30. app.post('/create', async (*req*,*res*)*=>*{
32. //fazendo um destruct e capturando as informações passadas via post
33. *var* {name, email, cpf, description, date, time} = *req*.body;

36. *var* status =   await appointmentService.Create(
37. name,
38. email,
39. cpf,
40. description,
41. date,
42. time
43. );
44. if(status){
45. //volta para página inicial
46. *res*.redirect('/');
47. }else{
48. *res*.send("Ocorreu uma falha!");
49. }
50. })
51. c
52. após isso vamos na página de cadastro create.ejs
53. vamos adicionar o method no formulário e o action.
54. <form method="post" action="create">
55. Rode o servidor entre na página de cadastro e faça um registro, se tudo ocorrer bem, o usuário será direcionado para a página principal .
56. Para verificar se o registro foi salvo no banco de dados entre no cmd digite:
57. Mongo e irá abrir a página de configuração do mongo
58. Show dbs irá abrir os bancos de dados salvos
59. Use agendamento irá abrir o banco de dados que foi criado
60. Show collections irá mostrar a tabela appointments
61. db.appointments.find() irá mostrar todos os registros salvos no banco de dados.
62. 

**SETUP DO FULL CALLENDAR**

1. Vamos criar a página inicial do sistema chamada index.ejs, onde terá o calendário com os apontamentos.
2. Neste arquivo iremos importar os formatos que foram carregados junto da biblioteca do fullcallendar.
3. Vamos criar uma div com um id chamado calendar onde o calendário será exibido.
4. No final da página vamos criar um script neste script vamos criar uma variável chamada elemento que irá receber a div de id calendar.
5. Vamos criar uma variável chamada calendar que receberá uma nova instancia de FullCalendar,
6. Esta instância possui um método chamado Calendar onde passamos como parâmetro a div element e um json com os parâmetros do callendar que neste caso é o formato da view, o idioma e o evento ‘getCalendar’
7. No final nós renderizamos o calendar para exibir o calendário na página.
8. <!DOCTYPE html>
9. <html lang="en">
10. <head>
11. <meta charset="UTF-8">
12. <meta http-equiv="X-UA-Compatible" content="IE=edge">
13. <meta name="viewport" content="width=device-width, initial-scale=1.0">
14. <title>Sistema de Agendamentos</title>
15. <link rel="stylesheet" href="/fullcallendar/main.css">
16. <script src="/fullcallendar/main.js"></script>
17. </head>
18. <body>
19. <div id="calendar"></div>
20. </body>
21. <script>
22. *var* element = document.getElementById('calendar');
23. *var* calendar = new FullCalendar.Calendar(element, {
24. initialView: 'dayGridMonth',
25. locale: 'pt-br',
26. events: 'getcalendar'
27. });
28. calendar.render();
29. </script>
30. </html>
31. Após isso vamos na rota principal do arquivo index.js e alteramos para que seja renderizada a página index.ejs.
32. //rota principal
33. app.get('/',(*req*, *res*)*=>*{
34. //res.send("Olá!");
35. *res*.render('index');
36. });
37. Feito isso basta entrar na rota principal e o calendário será exibido
38. 

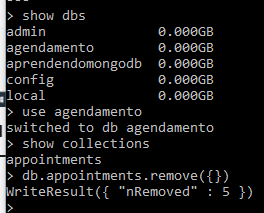
**METODO GetAll**

1. No servisse AppointmentService.js, vamos criar um método para exibir todos os registros do model AppModel.
2. Este método irá receber como parâmetro uma variável informando se a consulta irá mostrar todos os registros ou somente os registros que ainda não foram finalizados.
3. O método fará a consulta ao model retornando todos os registros caso o parâmetro seja true.
4. Caso o parâmetro seja false iremos criar uma variável appos que irá receber os resultado da busca de todos os registros do tipo false porém temos que melhorar o formato dos registros para visualização no calendário.
5. Vamos criar um array chamado appointments
6. Vamos realizar um forEach em appos e para cada registro encontrado nos iremos verificar se a data do registro foi definida, neste caso iremos adicionar este registro no array, porém vamos utilizar o método Build do factory AppointmentFactory que irá tratar as informações e salvar no array no formato adequado e no fim vamos retornar o array.
7. //listar todas as consultas recendo um booleano
8. async GetAll(*showFinished*){
9. //mostrando todas as consultas
10. if(*showFinished*){
11. return await AppoModel.find();
12. }else{
13. //mostrando as consultas que ainda não foram finalizadas.
14. //a variavel apps irá receber todos os registros que ainda
15. //não foram finalizados.
16. *var* appos = await AppoModel.find({'finished': false});
17. //array que receberá os registros com o formato ideal para
18. //mostrar no calendario.
19. *var* appointments = [];
20. //faremos um laço em cada registro
21. //e iremos adicionar
22. appos.forEach(*appointment* *=>* {
24. if(*appointment*.date != undefined){
25. appointments.push(AppointmentFactory.Build(*appointment*));
26. }
27. })
28. return appointments;
29. }
30. }
31. Finalizado o método GetAll, vamos no arquivo index.js e vamos criar uma rota para exibir todos os registros.
32. //rota para exibir todos os registros inclusive as consultas já finalizadas
33. app.get('/getcalendar',async (*req*,*res*)*=>*{
35. *var* appointments = await AppointmentService.GetAll(false);
36. *res*.json(appointments);
37. })
38. **Padrão de projeto Factory**
39. Precisamos alterar a data que será mostrada no fullcallendar, pois precisamos do dia e hora tudo junto. .
40. Crie uma pasta chamada factories e um arquivo chamado AppointmentFactory.js
41. Neste arquivo vamos criar uma classe chamada AppointmentFactory com um método chamado Build que irá receber uma consulta simples e irá retornar um json com alguns campos pré-determinados.
42. *class* AppointmentFactory{
43. Build(*simpleAppointment*){
44. //vamos juntar a hora e a data para isso vamos pegar partes da
45. //data e vamos juntálos
46. *var* day = *simpleAppointment*.date.getDate();
47. *var* month = *simpleAppointment*.date.getMonth();
48. *var* year = *simpleAppointment*.date.getFullYear();
50. //pegando a hora para isso vamos ter que fazer um split
51. //nos dois pontos sabendo que o primeiro elemento do
52. //array  são as horas e o segundo são os minutos;
53. //precisamos converter as informações de horário para número
54. *var* hour = *Number*.parseInt(*simpleAppointment*.time.split(":")[0]);
55. *var* minutes = *Number*.parseInt(*simpleAppointment*.time.split(':')[1]);
56. //criando a variavel de início da consulta, juntamos todas as
57. //variáveis incluindo os segundos e milisengundos que serão zero
58. *var* startDate = new *Date*(year, month, day, hour, minutes, 0, 0);
59. //alterando a hora para desconsider as 3 horas à mais da UTC
60. startDate.setHours(startDate.getHours() + 6  );
62. //criando um json com os novos campos para exibir no calendario
63. *var* appo = {
64. id: *simpleAppointment*. \_id,
65. title: *simpleAppointment*.name + " - "+ *simpleAppointment*.description,
66. start: startDate,
67. end: startDate
68. }
69. return appo;
70. }
71. }
72. *module*.*exports* = new AppointmentFactory();

**DETECÇÃO DE CLIQUE NO FULLCALENDAR**

1. No script da página index.ejs vamos alterar a variável calendar, incluindo o evento click com uma função que irá receber um parâmetro chamado info que é um objeto passado pelo event ‘getcalendar’.
2. Vamos utilizar o window.location.href para enviar o usuário para uma rota /event/:id com os dados do registro passado pelo info.id.
3. *var* element = document.getElementById('calendar');
4. *var* calendar = new FullCalendar.Calendar(element, {
5. initialView: 'dayGridMonth',
6. locale: 'pt-br',
7. events: 'getcalendar',
8. eventClick: *function*(*info*){
9. window.location.href = '/event/'+*info*.event.id;
10. }
11. });
12. calendar.render();
13. No arquivo index.js Vamos criar a rota para carregar a página do id passado pelo eventClick.
14. //rota para carregar um registro em uma página paa edição.
15. app.get('/event/:id', (*req*, *res*)*=>*{
16. *res*.json({id: *req*.params.id});
17. });

**CRIANDO MÉTODO GETBYID**

1. No arquivo servisse AppointmentService vamos criar um método para localizar um registro no mongo pelo id que foi passado via requisição no evento click da página do calendário.
2. async GetById(*id*){
3. try {
4. *var* event = await AppoModel.findOne({'\_id': *id*});
5. return event;
6. } catch (err) {
7. console.log(err);
8. }
10. }
11. No arquivo index.js na rota app.get(“event/:id) vamos criar uma variável appointment que irá receber o resultado do método GetById
12. Vamos renderizar a pagina event.ejs e vamos passar a variável appointment como um json.
13. //rota para carregar um registro em uma página paa edição.
14. app.get('/event/:id', async(*req*, *res*)*=>*{
15. /\*testando
16. console.log(await AppointmentService.GetById(req.params.id));
17. res.json({id: req.params.id});
18. \*/
19. *var* appointment = await AppointmentService.GetById(*req*.params.id);
20. *res*.render('event',{appo: appointment});
21. });
22. Vamos criar a página event.ejs que irá carregar o registro e um botão para finalizar a consulta.
23. <h1>Consulta: <%= appo.description %> </h1>
24. <hr>
25. <h4>Paciente: <%= appo.name %></h4>
26. <hr>
27. <h4>Email: <%= appo.email %> </h4>
28. <hr>
29. <h4>CPF: <%= appo.cpf %> </h4>
30. <hr>
31. <% if(!appo.finished){ %>
32. <button>Finalizar consulta</button>
33. <% } %>
34. **FINALIZANDO UMA CONSULTA**
35. Para a finalização de uma consulta, vamos criar um método no servisse AppointmentService, chamado appointmentFinished que receberá como parâmetro o id da consulta que será finalizada.
36. Neste método iremos utilizar o método ‘findByIdAndUpadate’ do mongo, para isso utilizaremos o model AppoModel, passando como parâmetros o id, o campo e o valor do campo que será alterado.
37. //classe para alterar o campo booleando de consulta finalizada
38. async appointmentFinished(*id*){
39. *var* msg;
40. try {
41. *var* result = await AppoModel.findByIdAndUpdate(*id*,{finished: true});
42. if(result){
43. console.log("Consulta finalizada");
44. msg = "Constulta finalizada";
45. return {status: true, msg: msg};
46. }else{
47. console.log("Consulta não encontrada");
48. msg = "Consulta não localizada";
49. return {status: false, msg: msg};
50. }
51. } catch (err) {
52. console.log("Erro ao finalizar consulta: "+ error);
53. return {status: false, msg: err};
54. }
55. }
56. Após isso vamos criar um formulário em volta do botão da página event.ejs, para podermos enviar para a rota ‘/finished’ o id da consulta que será finalizada.
57. <form  method="post" action="/finished">
58. <input type="hidden" name="id" value="<%= appo.\_id %>">
59. <button>Finalizar consulta</button>
60. </form>
61. Agora iremos criar a rota ‘/finished’ no arquivo index.js
62. Essa rota receberá o id no corpo da requisição, por isso criamos a variável id que receberá o id que foi passado no corpo da requisição.
63. Utilizamos então o AppointmentService que foi importado para a index.js e passamos para seu método appointmentFinished, que criamos anteriormente, o id da consulta que será finalizada, se der tudo certo a consulta será finalizada no banco de dados e o usuário será direcionado para página principal ‘index’ com o calendário já atualizado.
64. /rota para finalizar a consulta
65. app.post('/finished', async(*req*, *res*)*=>*{
66. *var* id = *req*.body.id;
67. *var* result = await AppointmentService.appointmentFinished(id);
68. *res*.redirect('/');
69. });
70. **LISTAGEM E BUSCA**
71. No arquivo index.js vamos criar uma rota chamada ‘/list’ que irá exibir todos os registros do método ‘GetAll’ do model AppointmentService.
72. Vamos renderizar uma view chamada list passando para ela o resultado da busca.
73. //rota para exibir dos os registros
74. app.get('/list',async (*req*,*res*)*=>*{
75. *var* appo = await AppointmentService.GetAll(true);
76. *res*.render('list',{appo});
77. })
78. **VIEW LIST**
79. Vamos criar uma nova view chamada list.ejs. nessa list view nos vamos criar um forEach com o ‘appo’ enviado pelo render. E neste forEach vamos listar cada item.
80. <h1>
81. Consultas:
82. </h1>
83. <hr>
84. <form method="post" action="">
85. <input type="search" name="search" id="search" placeholder="Digite o email ou CPF do paciente">
86. <button>Pesquisar</button>
87. </form>
88. <hr>
89. <% list.forEach(appo => { %>
90. <h3>Nome: <%= appo.name %></h4>
91. <h4>Descrição: <%= appo.description %></h4>
92. <h4>CPF: <%= appo.cpf %></h4>
93. <hr>
94. <% }); %>
95. **CRIANDO MÉTODO Search**
96. Vamos criar um método que realizará a busca no banco de dados utilizando o CPF ou o email do cliente, passados como parâmetro.
97. No arquivo AppointmentService vamos criar o método Serach que receberá um parâmetro que poderá ser o CPF ou o email de um paciente.
98. Iremos utilizar o método find().or(), onde dentro de or iremos informar as condições para a procura.
99. //metodo que irá localizar um registro através do cpf ou o email do usuário
100. async Search(*query*) {
101. try {
102. *var* result = await AppoModel.find().or([
103. {email: *query*},
104. {cpf: *query*}
105. ]);
106. if(result.length>0){
107. return result;
108. }else{
109. return {status: false, msg: "Paciente não localizado" };
110. }
111. } catch (err) {
112. return {status: false, msg: err};
113. }
114. }
115. **ROTA ‘/searchresult’**
116. Criamos a rota para carregar o resultado da busca na pagina list.
117. /rota para exibir a buscar por email ou cpf
118. app.get('/searchresult',async(*req*, *res*)*=>*{
119. *var* query = *req*.query.search;
120. *var* list = await AppointmentService.Search(query);
121. //res.json(search);
122. *res*.render('list',{list});
123. });
124. **NOTIFICAÇÃO DE CLIENTES COM SETINTERVAL**
125. Iremos implemtar uma maneira de notificar os clientes sobre a consulta, que ela está preste a acontecer.
126. No arquivo principal index.js vamos chamar o setInterval executando uma função a cada intervalo de tempo.
127. //setando um tempo para executar a consulta ao banco de dados
128. setInterval( async () *=>* {
129. console.log("Olá");
130. AppointmentService.sendNotification();
131. }, config.taskTime);
132. Iremos configurar uma consulta ao banco de dados para verificar se alguma consulta irá ocorrer na próxima hora.
133. Crie o arquivo config.js e configure uma variável de tempo para fazer as consultas ao banco de dados.
134. *module*.*exports* = {
135. //60000 corresponde a um minuto, para oter outro
136. //tempo basta multiplicar.
137. taskTime: (60000 \* 2)
138. }
139. C
140. Vamos alterar o model do banco de dados e criar um campo chamado notification que será true ou false, caso seja false, significa que o usuário ainda não foi notificado da consulta que irá ocorrer.
141. *onst* mongoose = require('mongoose');
142. *const* appointment = new mongoose.Schema({
143. name: *String*,
144. email: *String*,
145. cpf: *String*,
146. description: *String*,
147. date: *Date*,
148. time: *String*,
149. finished: *Boolean*,
150. notified: *Boolean*
151. });
152. module.exports = appointment;
153. vamos alterar também o AppointmentService incluindo o campo nofified ao método create.
154. async Create(*name*, *email*, *cpf*, *description*, *date*, *time*){
155. *var* newAppo = new AppoModel({
156. name: *name*,
157. email: *email*,
158. description: *description*,
159. cpf: *cpf*,
160. date: *date*,
161. time: *time*,
162. finished: false,
163. notified: false
164. });
165. Acesse o mongo e apaque todos os registros de testes.
166. ****
167. No arquivo AppointmentFactory vamos adicionar mais um campo ao json appo.
168. *var* appo = {
169. id: *simpleAppointment*. \_id,
170. title: *simpleAppointment*.name + " - "+ *simpleAppointment*.description,
171. start: startDate,
172. end: startDate,
173. notified: *simpleAppointment*.notified
174. }

**METODO SendNotification**

1. No arquivo AppointmentService vamos criar um método chamado SendNotification neste método vamos criar a variável appos que receberá o resultado da busca do método GetAll. Iremos fazer um forEach. Dentro do forEach vamos criar uma variável date que irá receber a data de e hora da consulta. Vamos criar também uma variável chamada ‘hour’ que terá a quantidade de horas que antecede a consulta. Criamos também a variável ‘gap’ que será a diferença entre a data atual e a data da consulta, se esta diferença for menor ou igual à ‘hour’ será realizado a rotina de envio de email para o paciente.
2. //metodo para enviar a notificação ao paciente
3. async sendNotification(){
4. *var* appos = await this.GetAll(false);
5. *var* time = *Date*.now();
7. appos.forEach(*appo* *=>*{
8. //date recebe a data do banco de dados
9. *var* date = *appo*.start.getTime();
10. //corresponde ao tempo de antecedência para envio da notificação
11. *var* hour = 1000 \* 60 \* 60 \* 1;
12. //gap é diferença entre a data do sistema menos a data atual
13. *var* gap = date - *Date*.now();
14. if(gap <= hour){
15. console.log(*appo*.title);
16. console.log("Mand a not!");
17. }
18. });

**ENVIO DE EMAIL**

1. Primeiramente vamos configurar o envio de email utilizando o nodemailer.
2. Instale o nodemailer, digite npm install nodemailer –save,
3. Crie uma pasta config e dentro dela coloque o arquivo config.js. Neste arquivo configure o host, port, secure, user e pass que serão utilizados no arquivo de envio de email.
4. *odule*.*exports* = {
5. //60000 corresponde a um minuto, para oter outro
6. //tempo basta multiplicar.
7. taskTime: (3000 \* 10),
8. host: "smtp.gmail.com",
9. //TSL EXIGIDO
10. //port: 587,
11. //SSL EXIGIDO
12. port: 465,
13. secure: true,
14. user: "testdevclaudisnei@gmail.com",
15. //pass: "senha\_do\_email"
16. pass:"senha\_do\_email"
17. }
18. crie outro arquivo chamado email.js
19. No arquivo email.js importe o nodemailer.
20. Importe também as configurações do arquivo config.js
21. Crie uma variável transporter que irá utilizar o método createTransport do nodemailer e configure as informações de onde o email será enviado, como host, port, secure, user, pass.
22. Utilize o método sendMail do transporter para preencher os dados do email que será enviado como, from, to, subject, text e html
23. Esse método é uma promisse então configure uma mensagem e um err.
24. *const* nodemailer = require('nodemailer');
25. //importando configuraçãoes
26. *const* config = require('./config');
27. //criando um transporting
28. *let* transporter = nodemailer.createTransport({
29. host: config.host,
30. port: config.port,
31. secure: config.secure,
32. auth:{
33. user: config.user,
34. pass: config.pass
35. }
36. });
37. //criando as configurações do email que será utilizado
38. transporter.sendMail({
39. from: "Claudisnei <testdevclaudisnei@gmail.com>",
40. to: "claudisneibello@gmail.com",
41. subject: "Oi estou configurando o envio de email 2",
42. text:"Esta é a mensagem que iremos passar no email para o usuário",
43. html:"Olá eu sou Claudisnei estou aprendendendo o <a href='https://guiadoprogramador.com'>nodemaile</a> é muito legal"
44. }).then(*message* *=>*{
45. console.log(*message*);
46. }).catch(*err* *=>*{
47. console.log(*err*);
48. })
49. **MAILTRAP**
50. Iremos utilizar os serviços do mailtrap para simular o recebimento dos e-mails enviados, desta forma poderemos ver se os email estão sendo recebidos pelo usuário.
51. Vá no site https://mailtrap.io
52. Crie uma conta, entre no My Inbox, verifique as configurações de SMTP Settings e copie os parâmetros de configuração host, port, user e pass para configurarmos o arquivo config.js com estes parâmetros.
53. Após preencher as configurações de config, vamos configurar o método sendNotification do arquivo AppointmentService.
54. //metodo para enviar a notificação ao paciente
55. async sendNotification(){
56. *var* appos = await this.GetAll(false);
57. // var time = Date.now();
58. //criação da configuração do nodemailer para envio de email
59. *var* transporter = mailer.createTransport({
60. host: config.host,
61. port: config.port,
62. auth:{
63. user: config.user,
64. pass: config.pass
65. }
66. });
67. appos.forEach(async *appo* *=>*{
68. //date recebe a data do banco de dados
69. *var* date = *appo*.start.getTime();
70. //corresponde ao tempo de antecedência para envio da notificação
71. *var* hour = 1000 \* 60 \* 60 \* 1;
72. //gap é diferença entre a data do sistema menos a data atual
73. *var* gap = date - *Date*.now();
74. if(gap <= hour){
75. //console.log(appo.title);
76. //console.log("Mand a not!");
77. //iremos verificar se o paciente já foi notificado
78. if(!*appo*.notified){
80. await AppoModel.findByIdAndUpdate(*appo*.id, {notified: true});
81. //envio de email
82. transporter.sendMail({
83. from:"Claudisnei <testdevclaudisnei@gmail.com",
84. to: *appo*.email,
85. subject: "Alerta do horário da consulta",
86. text: "A sua consulta está marcada e irá acontecer em 1h"
87. }).then(()*=>*{
88. console.log("Email enviado para " + *appo*.email);
89. }).catch(*err=>*{
90. console.log(*err*);
91. })
92. }
93. }
94. });
95. //console.log(time);
96. //   console.log(appos);
97. }