ROTEIRO PARA CRIAR AUTENTICAÇAO DE UMA API REST COM JWT

1. Criar sistema de login da API
2. NA pasta APIDEGAMES
3. Abrir o arquivo index.js
4. Inserir no banco de dados “uma tabela de usuários” para realizar a autenticação
5. users:[
6. {
7. id:1,
8. email:"adm@adm.com.br",
9. nome:"admin",
10. senha:"12345"
11. },
12. {
13. id:2,
14. email:"claudisnei@teste.com",
15. nome:"Claudisnei",
16. senha:"teste123"
17. }
18. ]
19. Para gerar o token temos que instalar a biblioteca jsonwebtoken
20. ‘npm install –save jsonwebtoken’
21. Criar uma variável para instanciar a jwt
22. Criar uma chave secreta que será utilizada para gerar o token
23. //instanciando a biblioteca jsonwebtoken
24. const jwt = require("jsonwebtoken");
25. //criando uma chave secreta para o jwt
26. const secretKeyJwt = "ffhhae@#$¨%fdd!@#4";
27. Gerar o token
28. **Criar uma rota para autencicação ‘/auth’ a autenticação do JWT utiliza um token. A rota pega o email passado por requisição e verifica se o email está cadastrado no banco de dados.**
29. Para gerar o token criamos a rota “app.post(/auth,” que salva os dados do usuário, id, email e senha que foram passados via post . Então é realizado uma verificação se a variável email foi definida se foi definida, é realizado uma busca no banco de dados para ver se existe algum usuário com o email cadastrado. Neste caso é realizado outra verificação se a senha fornecida é a mesma cadastrada no banco de dados. Se a senha estiver correta é utilizado a função ‘sign’ do JWT, nesta função devemos Incluir as informações que se quer carregar no token, aqui estamos passando três JSON que são: o id e email do usuário, a chave secreta que criamos, e o tempo de duração do token. Esse token será enviado como resposta no formato de Json,
30. /\*\*rota para autenticação \*/
31. app.post("/auth",(*req*,*res*)=>{
32. var {email, password} = *req*.body;
33. //validações
34. if(email != undefined){
35. //Verificando se o usuário é cadastrado
36. var user = DB.users.find(*us* => *us*.email == email);
37. if(user != undefined){
39. if(user.password === password){
40. /\*\*deve-se passar as informações necessárias
41. \* , a chave secreta e o prazo que irá expirar
42. \* o token
43. \*/
45. jwt.sign(
46. {
47. id: user.id,
48. email: user.email,
50. },
51. secretKeyJwt,{expiresIn:"48h"},(*err*, *token*)=>{
52. if(*err*){
53. *res*.status(400);
54. *res*.json({err: "Falha interna"});
55. }else{
56. *res*.status(200);
57. *res*.json({token: *token*});
58. }
59. })
60. }else{
61. *res*.status(401).json( "Email ou senha inválidos");
62. // res.json({err: "Email ou senha inválidos"});
63. }
64. }else{
65. *res*.status(404);
66. *res*.json({err: "Email ou senha inválidos"});
67. }
68. }else{
69. *res*.status(400);
70. *res*.json({err: "O email enviado é inválido."});
71. }
72. });
73. Protegendo as rotas através dos tokens
74. Vamos trabalhar com midlewares
75. Vamos criar um midleware
76. A função auth() captura o valor token que está salvo no header : authorization, caso não tenha token neste header, significa que o usuário não está cadastrado ou ainda não logou. Se tiver o token ele faz um split deste valor para retirar a palavra ‘bearer’ e gravar somente os caracteres do token. O jwt através do método verify recebe os valores dos parâmetros que foram criados na função jwt.sign lá na rota de autenticação ‘/auth’.
77. Voltando à função jwt.verigy(), os parâmetros que esta função recebe são: o token, a palavra secreta, e um array que irá conter o id e email do usuário. Se o resultado da verificação for o mesmo que foi criado na função sign, será enviado para a requisição um json contendo o id e email do usuário. E será permitido a aplicação passar adiante através do next(). Caso o token não corresponda ao que foi gerado será enviado uma mensagem de token inválido.
78. //midleware para realizar a autenticação antes da rota ser executada
79. function auth(*req*, *res*, *next*){
80. //pegando o header de autorização
81. const authToken = *req*.headers['authorization']
82. //validando o token
83. if(authToken !=undefined){
84. //cortando o token
85. const bearer = authToken.split(' ');
86. //pegando somente o token
87. var token = bearer[1];
88. //verificando se o token é valido
89. jwt.verify(token,secretKeyJwt,(*err*, *data*)=>{
90. if(*err*){
91. *res*.status(401);
92. *res*.json({err:"Token inválido!"});
93. }else{
94. // caso o token seja valido.
95. //passando variáveis para a requisição
96. //, estes dados poderão ser acessados
97. //nas rotas que forem autenticadas
98. *req*.token = token;
99. *req*.logedUser = {id: *data*.id, email: *data*.email};
100. //console.log(*data*);
101. next();
102. }
103. });
104. // console.log(bearer);
105. }else{
106. *res*.status(401);
107. *res*.json({err:"Token inválido"});
108. }
109. //console.log(authToken);
110. //se autenticação foi realizada com sucesso a função next que irá
111. //realizar a requisição do usuário
113. }
114. Incluir nas rotas o midleware para realizar a autenticação
115. Ex: app.get('/games',auth,(*req*,*res*)=>{
116. *res*.statusCode = 200;
117. *res*.json(DB.games);
118. });
119. Onde auth é o midleware
120. Incluir em todas as rotas o middleware para realizar a verificação se o usuário está logado na API
121. \*\*rota para devolver ao usuário o game pelo ID \*/
122. app.get('/game/:id',auth,(*req*,*res*)=>{
123. /\*\*novo endpoint para cadastrar um novo game \*/
124. app.post('/game',auth,(*req*,*res*)=>{
125. /\*\*novo endpoint para deletar um jogo \*/
126. app.delete('/game/:id',auth,(*req*,*res*)=>{
127. \*\*endpois para edição de registros será criado uma rota
128. \* utilizando o PUT
129. \*/
130. app.put('/game/:id',auth,(*req*,*res*)=>{