Relatório – Desenvolvimento da API + apache QA – 15/03/2023

Nome: Bruno do Nascimento Rodrigues

Git: https://github.com/Brunonasccimentto/client\_api

1- Objetivos

Desenvolver uma API node.js que conecta a um banco de dados mysql utilizando o método de requisição http “POST”, com as seguintes funções: buscar os tipos de cliente, registrar os dados enviados no banco de dados e consultar os dados baseado no id ou email do cliente.

2- Atividade

Com a prática da atividade anterior, já tinha conhecimento de quais pacotes usaria na aplicação node para conectar a um banco de dados mysql. Então, com a pasta criada vamos iniciar o node.

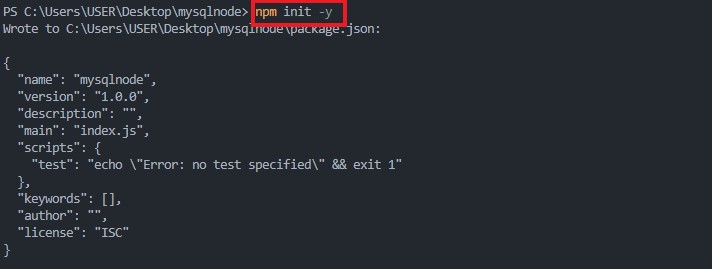


Figura 1 terminal após rodar npm init -y

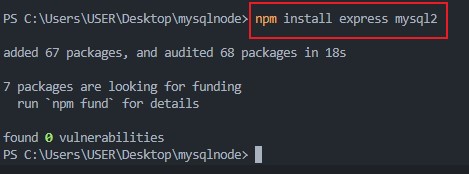


Figura 2 terminal após rodar npm install express mysql2

Primeiro foi criado uma estrutura básica para a api está ouvindo a porta 3000. Em vermelho “express.urlencoded”, para dizer que toda as requisições que esperamos seja como “form-urlencoded”, ou seja os dados do formulário são enviados em um único bloco. E a const “formRouter” que funciona como um middleware para manipular nossas rotas.

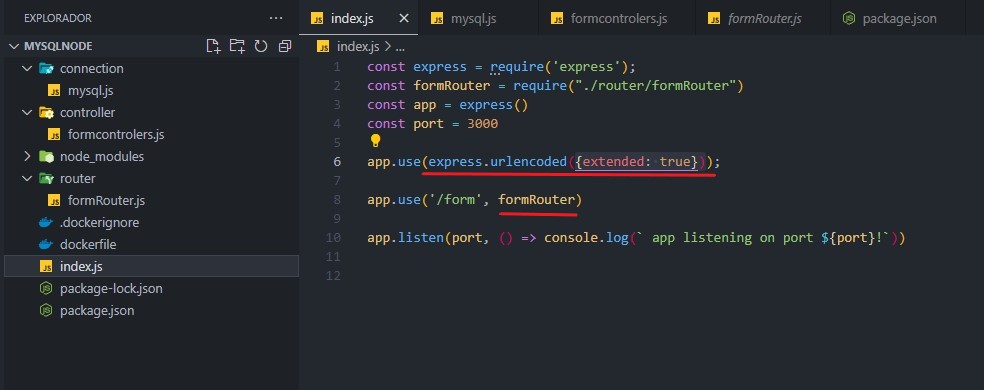


Figura 3 estrutura do arquivo index.js

Como a conexão ao banco de dados foi feita anteriormente, bastou copiar a mesma estrutura e somente mudar as configurações da conexão.



Figura 4 conexão ao mysql

Em primeiro momento, foi utilizado o mesmo ip como host e um outro database “db” para fins de teste antes de criar um container. Para esse código, foi criado uma pasta connection, e um arquivo de nome mysql.js.

Após criar a pasta router, criamos o arquivo formRouter.js, é neste arquivo que vai ficar todas as nossas rotas. Em amarelo, a const “router” chama a função Router() de express, que foi requerido logo acima, dessa forma podemos usar “router.post” para definir nossas rotas e a função que será executada ao chamá-la – A função só foi colocada após o arquivo “formControlers” está pronto.

Lembre-se de sempre exportar a const para usá-la em outro arquivo, no caso, no index.js.

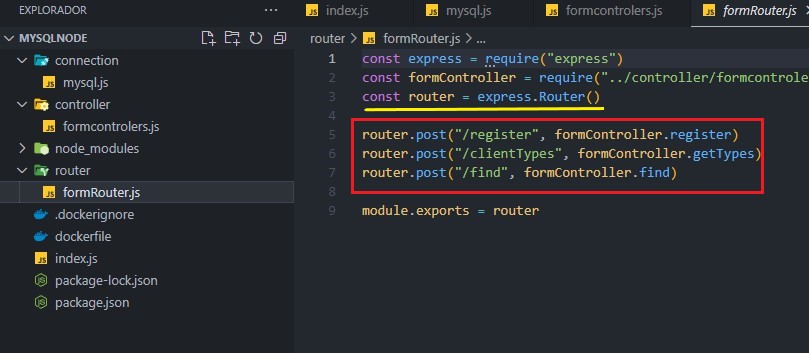


Figura 5 estrutura do arquivo formrouter.js

Por fim, criamos a pasta “controller”, com o arquivo “formControlers.js”. Neste arquivo estão localizadas nossas funções que serão usadas em “formRouter.js”.

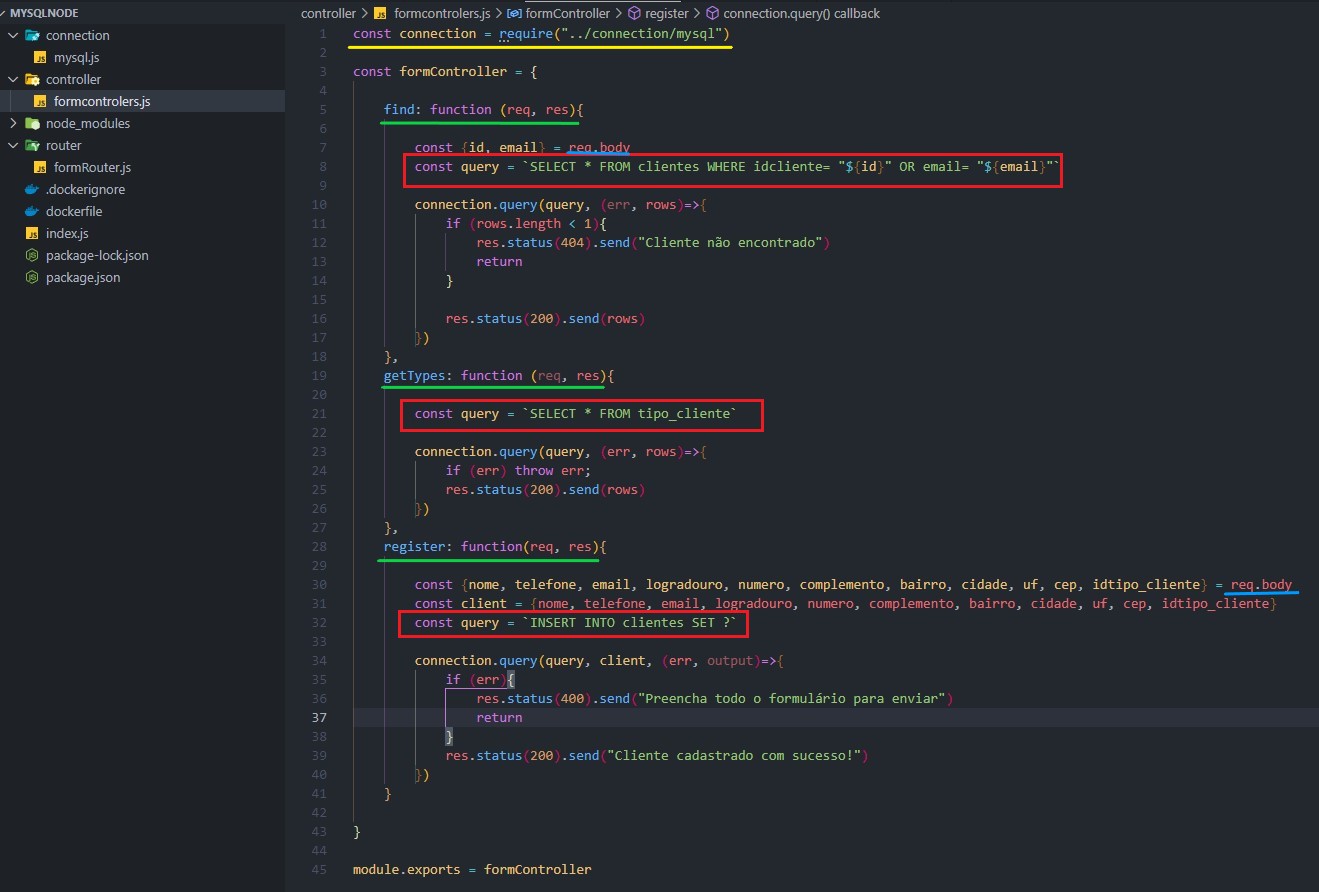


Figura 6 arquivo formControlers.js

Observe que em amarelo, estamos requerindo a connection do arquivo mysql da pasta connection. Em verde são os methods do objeto que criamos, “const formController”, cada method tem uma função que será executada para cada rota ao ser feito uma requisição. Em vermelho, são as queries em sql para retornarem o que desejamos ao serem chamadas. Na primeira query, observe o uso de “” ao passar ${} dentro de uma template literals, elas são essenciais para a query não falhar.

Em azul, agora usamos req.body a req.params, isso porque a requisição será feita pelo método http POST, ou seja, os parâmetros viram pelo corpo e não mais pela url. Vale lembrar que na hora da requisição, os nomes devem ser os mesmo das constantes, caso sejam diferentes virá o valor como “undefined”.

Agora vamos criar o banco que iremos trabalhar com essa api. Após rodar o container do mysql, entramos no Dbeaver e criamos um novo database de nome “db”, com as seguintes tabelas: clientes e tipo de clientes.

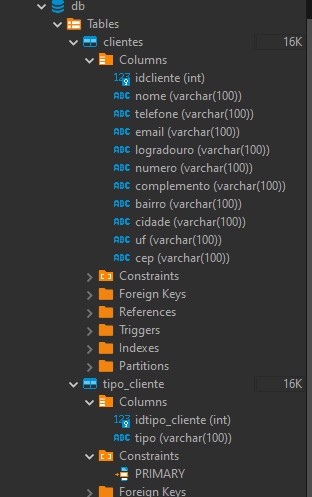


Figura 7 organização das tabelas clientes e tipo\_cliente

Com os respectivos dados:

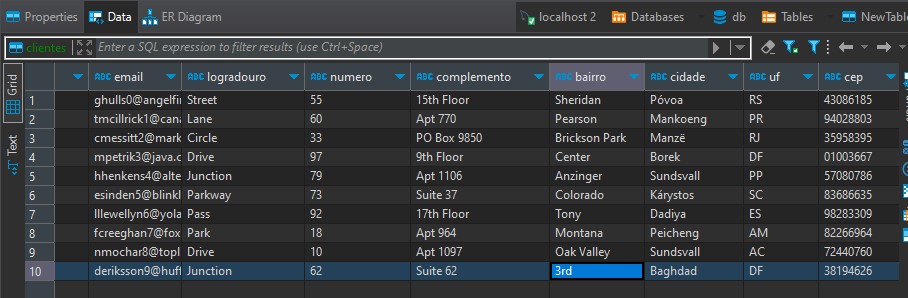


Figura 8 dados inseridos na tabela clientes.

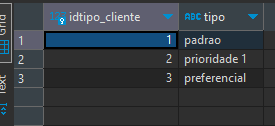


Figura 9 dados inseridos na tabela tipo\_cliente.

Com tudo finalizado, foi feito alguns testes no Insomnia para ver se a response vinha da forma esperada. - Testes foram feitos nas 3 rotas, porém não foi tirado todos os prints dessas etapas.

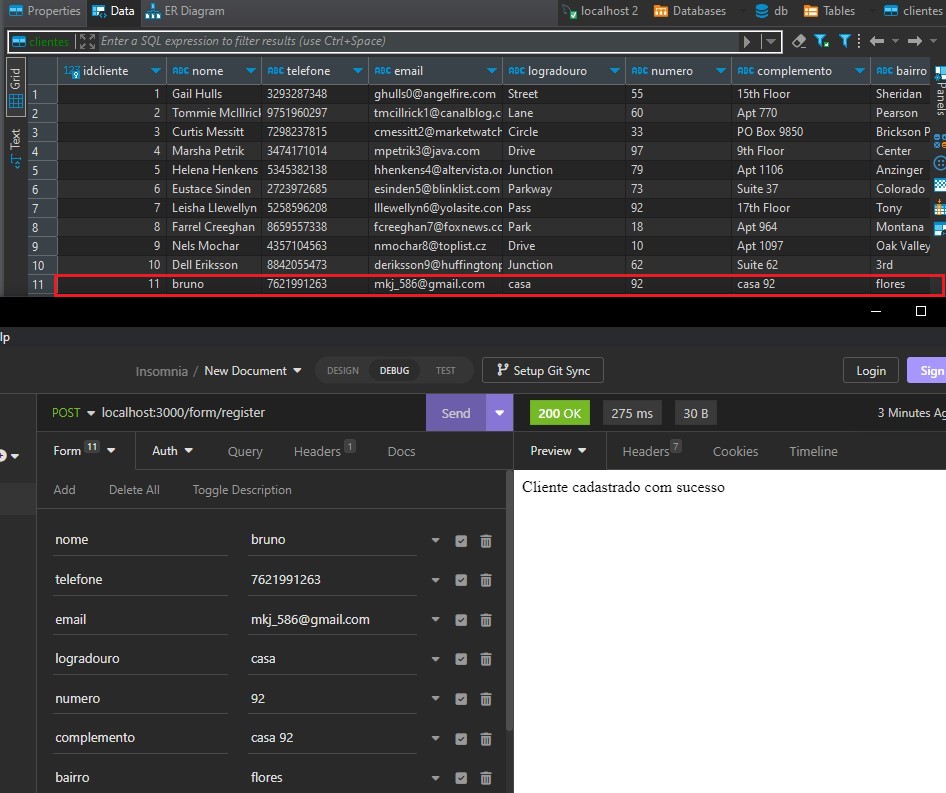


Figura 10 teste da api rota register.

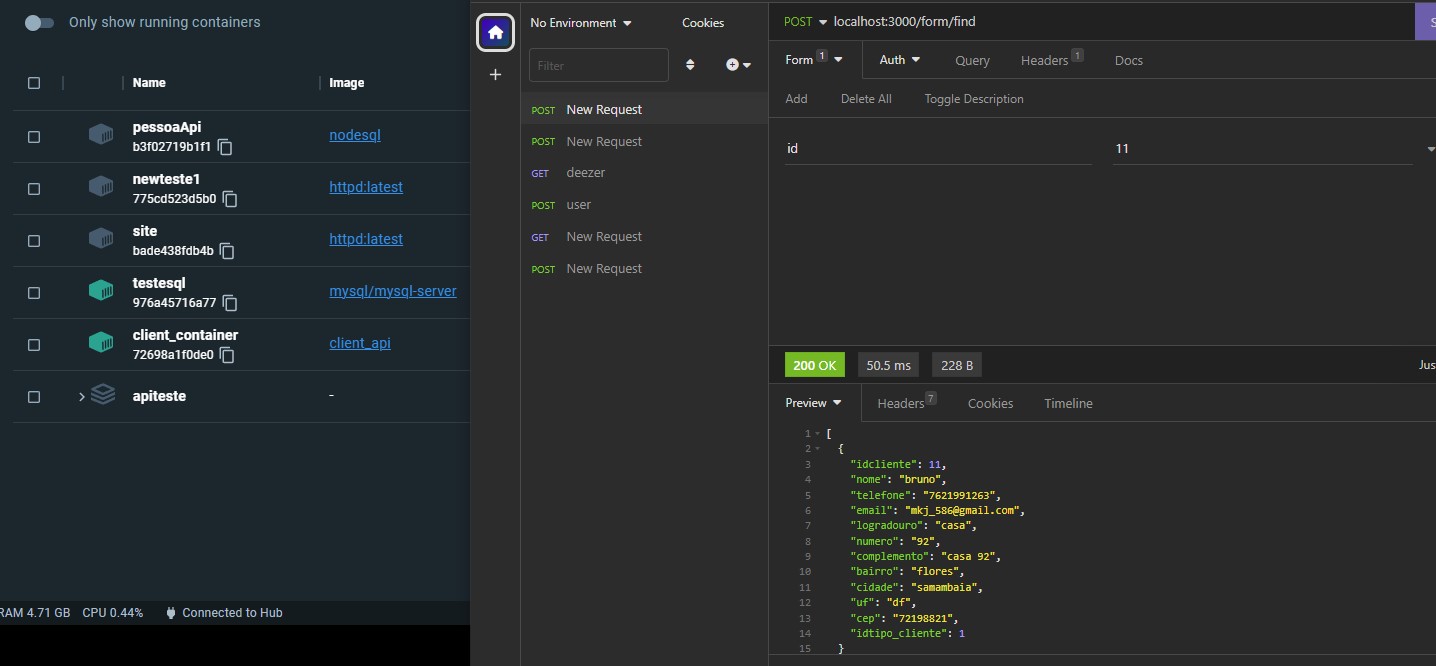


Figura 11 teste da api rota find feito após a criação do container.

Com tudo funcionando, só falta os arquivos dockerfile e .dockerignore.

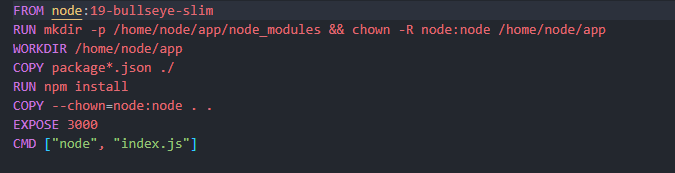


Figura 12 configuração do dockerfile

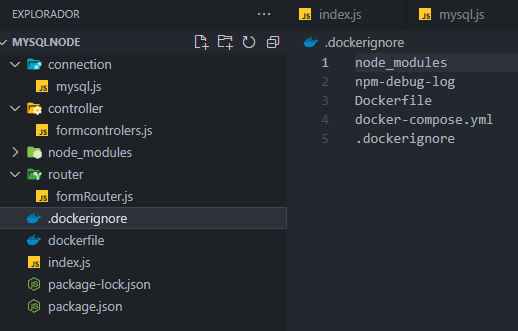


Figura 13 configuração do .dockerignore

Finalmente vamos subir para o docker. Em vermelho são os comandos para criar uma imagem dessa api com base nas configurações do dockerfile e criar um container respectivamente. Vale salientar que, --link é usado para “linkar” um container a outro, e é o que vamos fazer, “linkar” o container dessa api ao container do banco de dados.

Lembre-se de mudar o host da conexão no arquivo mysql.js, usaremos agora o nome do container.

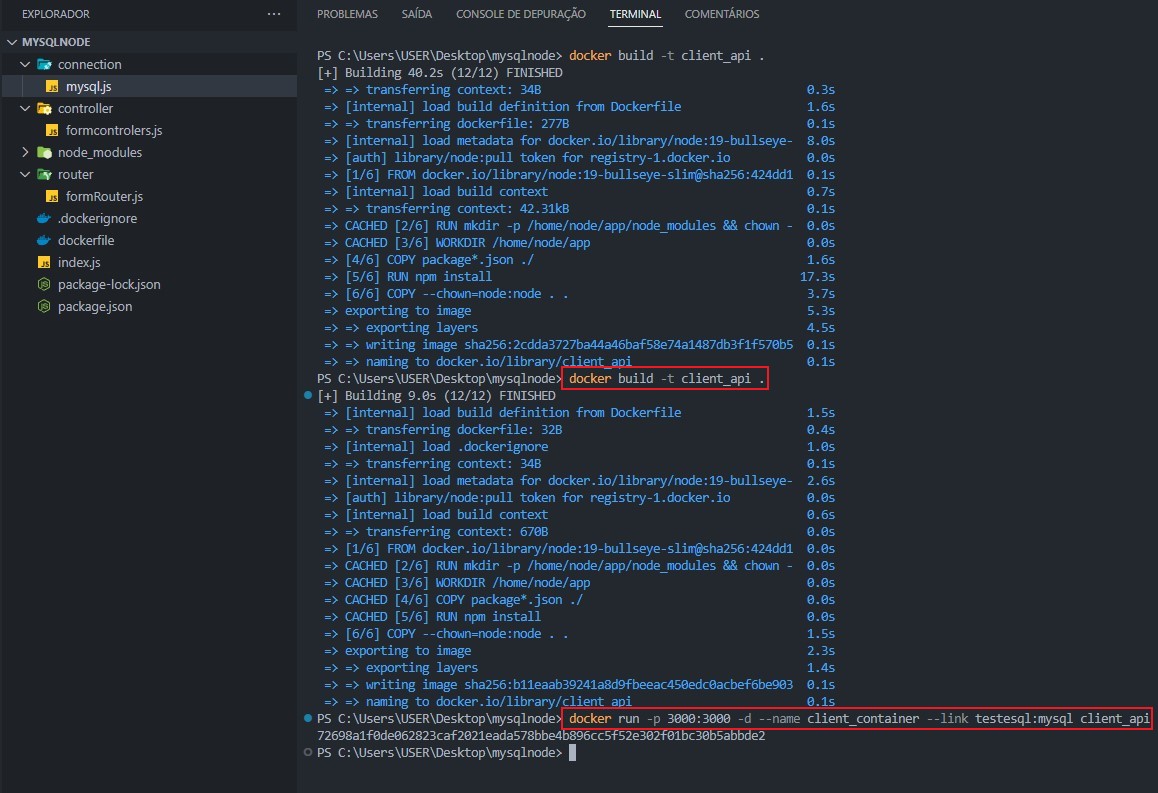


Figura 14 terminal após rodar os comandos

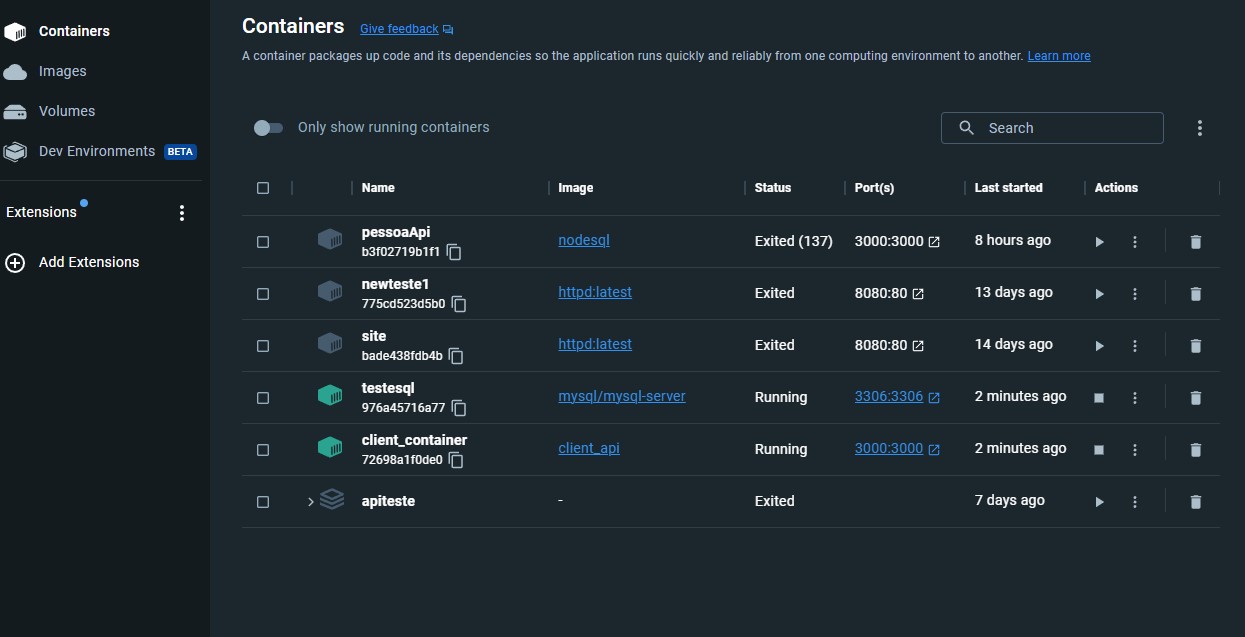


Figura 15 container rodando no docker.

3.1- Dificuldades e problemas encontrados - Api

O primeiro problema encontrado foi como fazer um registro no banco de dados, eu imaginava que teria que criar um schema com os dados esperados e seus respectivos tipos, porém, ao realizar uma pesquisa, descobri que era na verdade bem simples, era só usar query como já feito outras vezes.



Figura 16 site: <https://www.golinuxcloud.com/nodejs-save-form-data-in-mysql-table/>

Como eu não queria criar um if ou uma nova rota para tratar da busca por id e email, fiz uma busca para ver se existia um comando SQL que funcionasse como “ou”, e rapidamente foi esclarecida minha dúvida e arranjado uma solução.

O problema foi percebido depois de ter subido o container para o docker, a api não retornava uma resposta para email na query do “formController.find”.

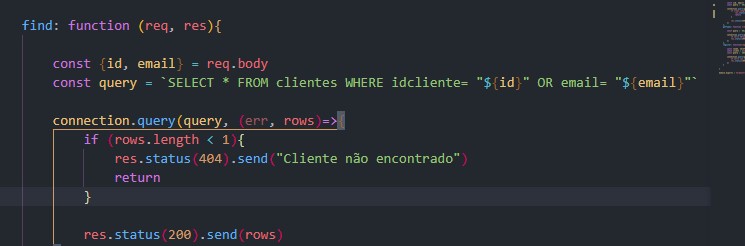


Figura 17 imagem da query já funcional

Fiz alguns testes com console.log(id, email) para ver o que retornava para mim no terminal, as vezes vinha e outras não o que era passado no insomnia. Depois de algumas horas percebi que se ${} não estiverem entres aspas, a query falha, sinceramente não sei o motivo, visto que fiz a query primeiro no script sql do dBeaver para testa-lá, e funcionava perfeitamente, porém, escrevi o id sem “”, visto que a comparação WHERE, era feita sobre dois números int.

Outro problema que so foi percebido posteriormente foi a necessidade do return em bad requests “404”, sem um return a api quebrava.



Figura 18 erro ao passar uma bad request.

3.2- Dificuldades e problemas encontrados - Docker

Com tudo funcionando perfeitamente, comecei a pensar mais sobre o docker e como iria passar o banco de dados para QA. Fazer um dump ou criar um novo banco e inserir os dados no pc de QA daria certo, porém não faz sentido no mundo real alguém que for utilizar esse banco ter que fazer toda uma configuração da imagem mysql/mysql-server, caso não tivesse um container com mysql, ou ainda ter que fazer o import do dumb. Isso demoraria alguns bons minutos todo esse processo.

Fiz muitas buscar para achar algo que resolveria esse problema, quase um dia inteiro pesquisando e dando erros, até que fiz a coisa mais óbvia, usar docker –help e ler os comandos.

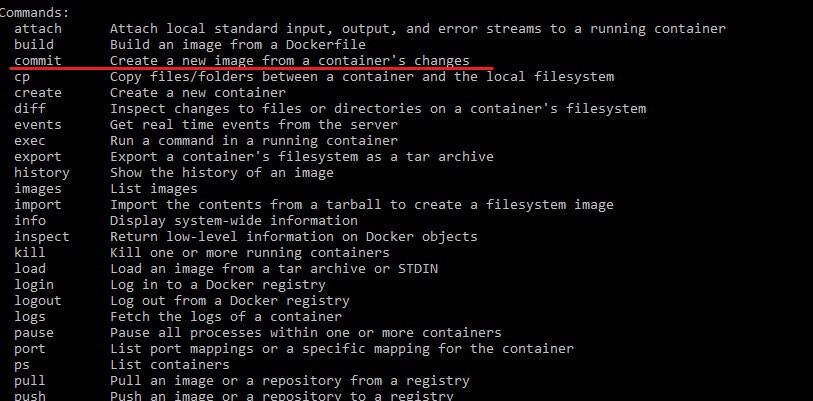


Figura 19 docker –help no cmd.

Quando eu vi commit e o que ele fazia, sabia que iria resolver o meu problema de levar o banco de dados. Dessa forma, busquei mais sobre o commit e encontrei esse site: <https://www.dataset.com/blog/create-docker-image/> ensinando a criar uma imagem a partir do container.

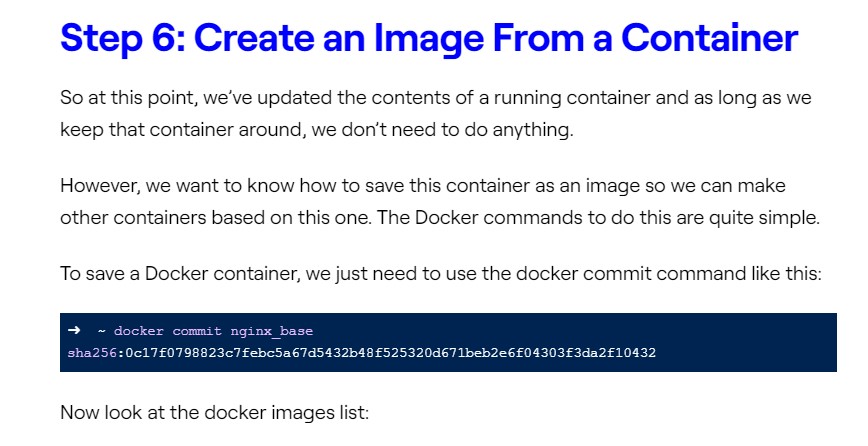


Figura 20 imagem do site acima.

Com isso voltei ao cmd e comecei com o commit



Figura 21terminal cmd apos rodar o commit

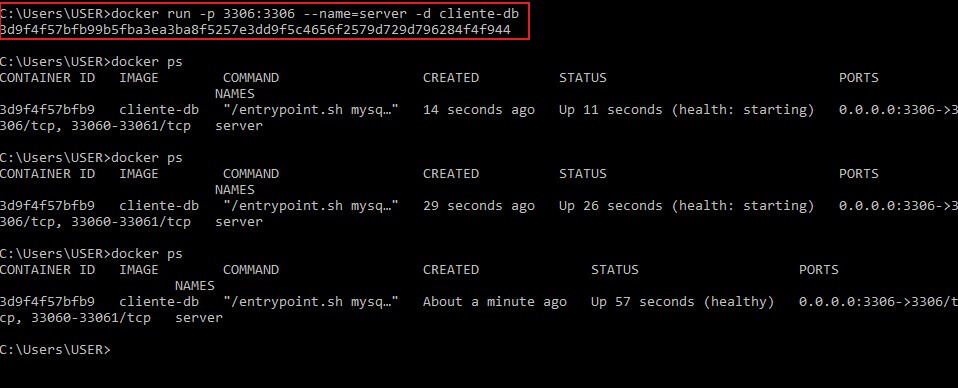


Figura 22 terminal cmd após criar um container com o id da imagem passada

Observe que não precisa configurar senhas nem criar usuários, e ainda rapidamente o status muda para (helthy), que significa que já pode está utilizando o container.

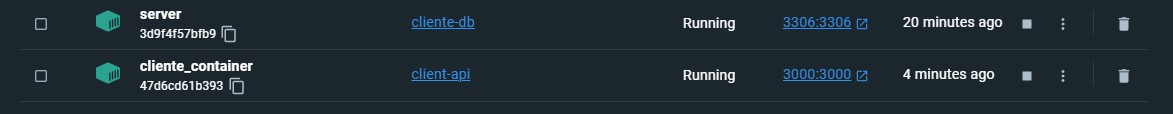


Figura 23 conteiner em execução com a imagem

Por final, fiz uma pesquisa de como subir uma imagem para o meu dockerhub, e encontrei o site <https://medium.com/codex/push-docker-image-to-docker-hub-acc978c76ad> que facilitou essa tarefa.

Primeiro criei um repositório no dockerhub e fiz a tag da minha imagem no cmd, passando o repositório como argumento. Com o tag feito, bastou passar o caminho do repositório no push.

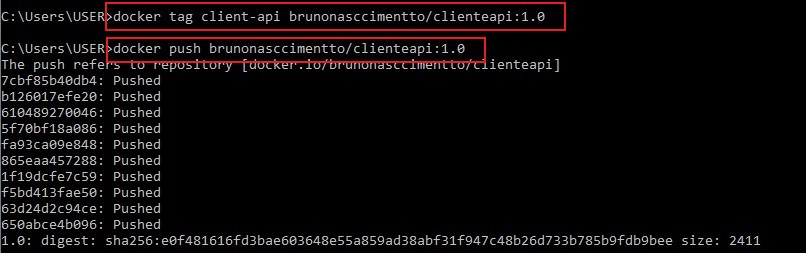


Figura 24 terminal cmd após subir uma imagem para o dockerhub

Após subir a api, repeti o mesmo processo para a imagem que contém o banco de dados.

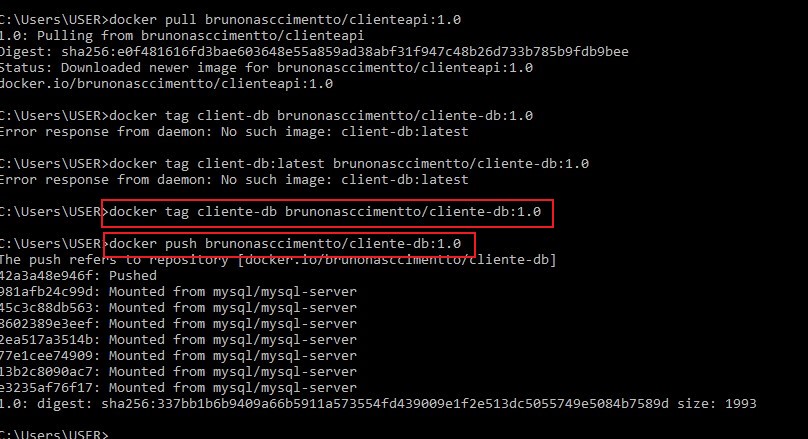


Figura 25 terminal cmd após subir a imagem com o db para o dockerhub

E assim ficou o repositório com as duas imagens:

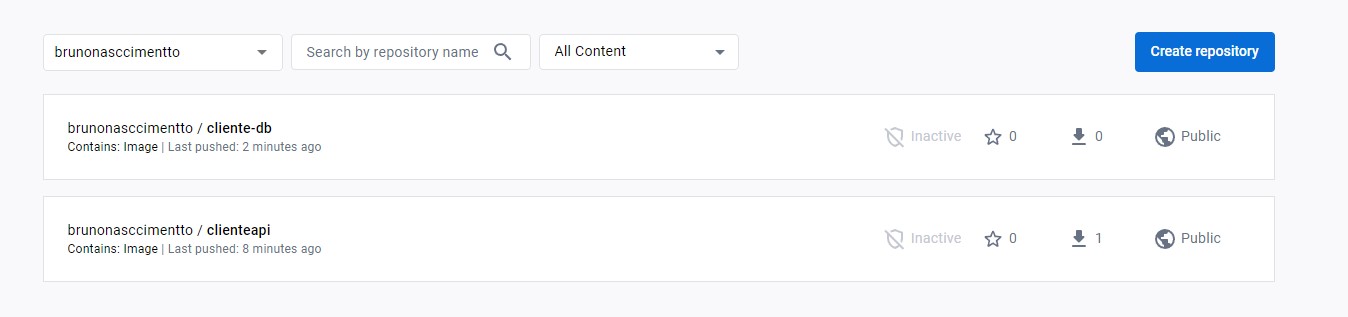
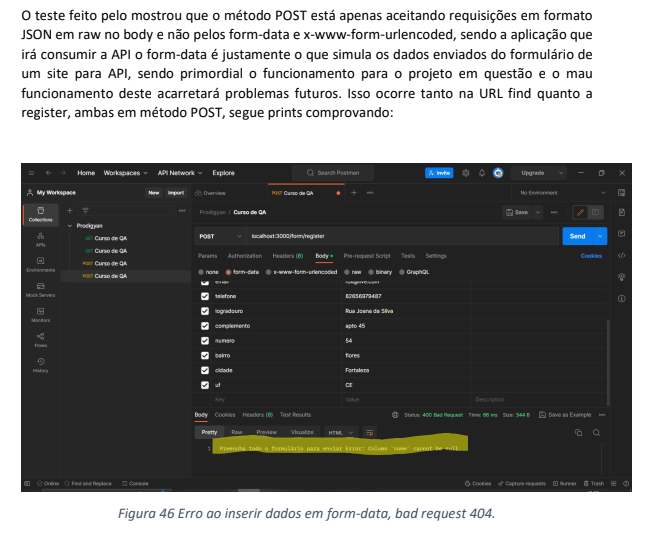


Figura 26 repositório dockerhub

3.3- Correções exigidas por QA – 25/03/23 as 20h



Foram necessárias pequenas mudanças no código para atender as exigências solicitadas. Primeiramente foi instalado o pacote “multer” para utilizar o formato exigido “multipart/form-data”. Após a instalação do pacote, colocado em uma constate para que seja utilizado como middleware na linha 11.

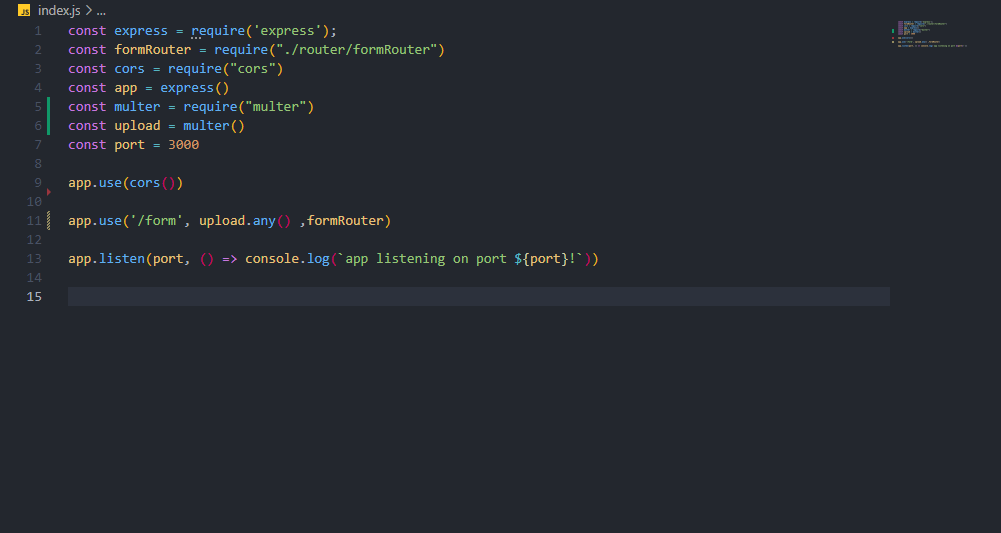
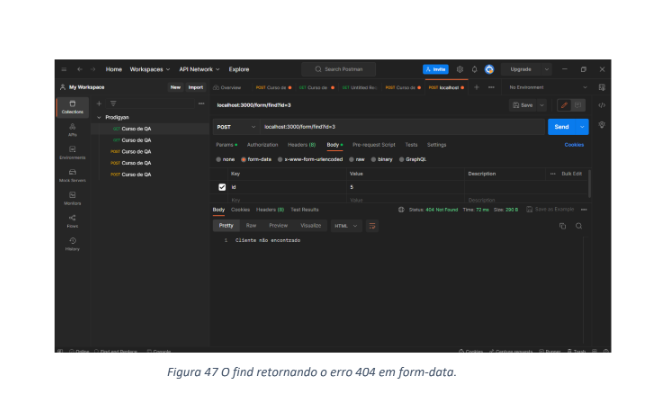


Figura 27 Mudanças realizadas no código



Outra exigência foi a mudança do método do caminho “/find” de POST para GET, para atender as exigências da atividade, onde no exemplo os dados de busca são passados na url.

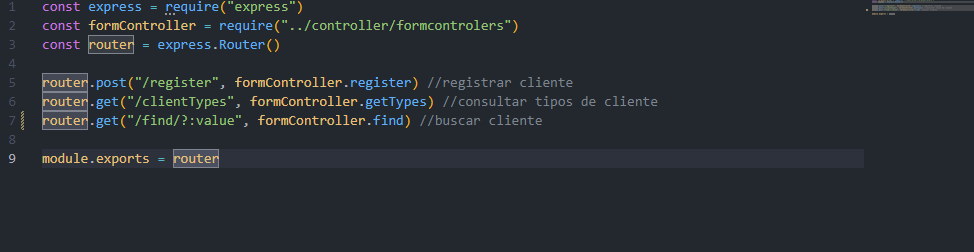


Figura 28 mudanças no método e rota do buscar clientes

Ficando assim as mudanças na função:

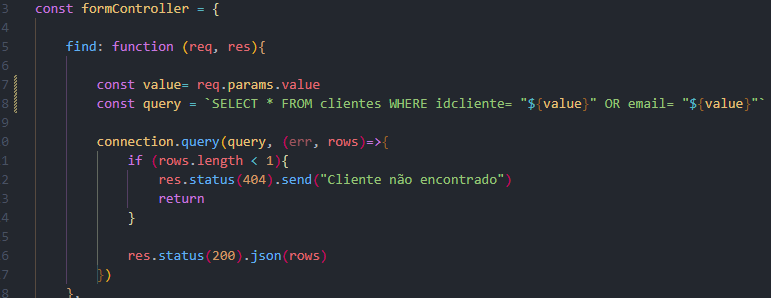


Figura 29 Alterações feitas na função find

Testes feitos comprovando o pleno funcionamento das mudanças:

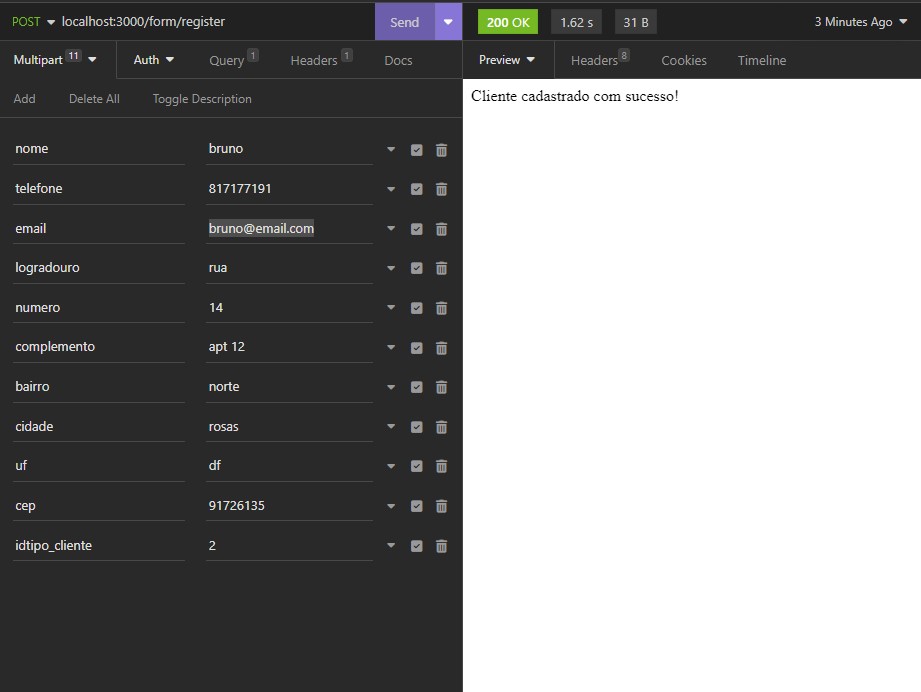


Figura 30 teste de registro utilizando multipart/form-data

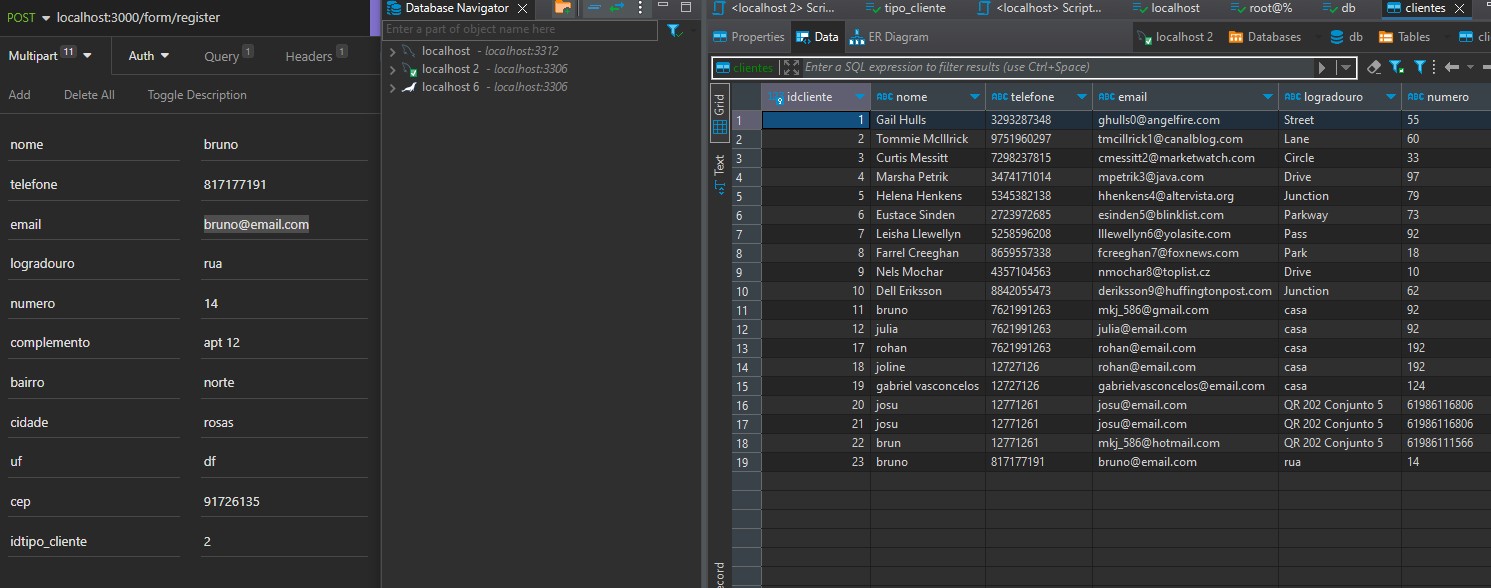


Figura 31 comprovando que os dados foram inseridos com sucesso no banco de dados

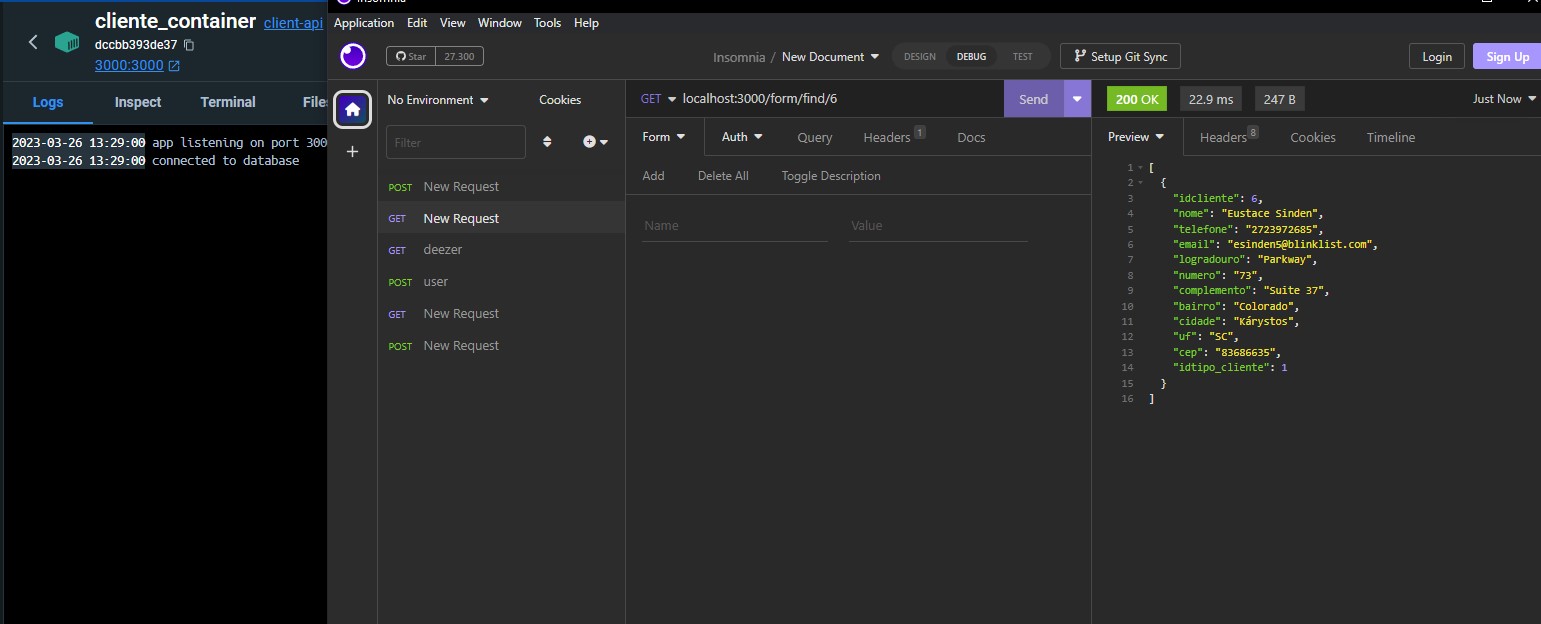


Figura 32 busca por id utilizando o método GET

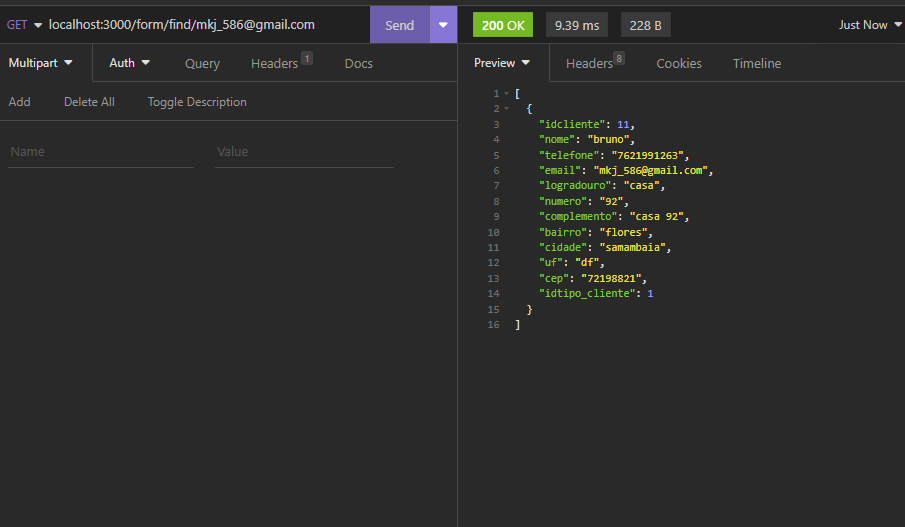
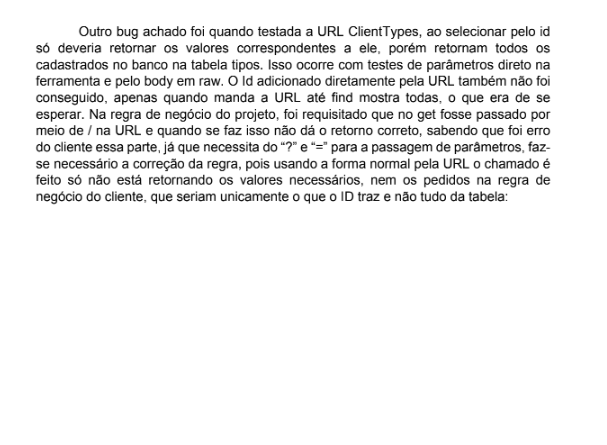
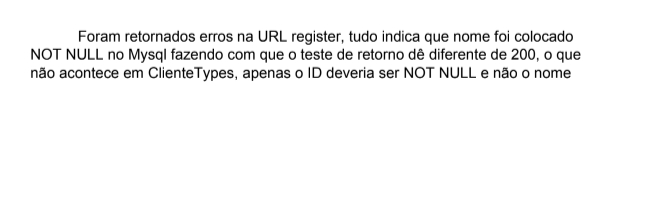


Figura 33 busca por email utilizando o método GET

Outros erros relatados:



Não foi realizada a mudança pois a atividade não pede em nenhum momento buscar por um tipo de cliente por id ou por tipo, somente a busca do cliente.



Não faz sentido que o banco de dados permita o registro de dados nulos na tabela cliente. Mesmo que o formulário já faça uma verificação se os dados são nulos, é importante não os permitir de maneira alguma.

3.4- Ultimas atualizações

Devido a uma explicação de que a busca deveria ser feita por query params e não params dentro da url, foram realizadas as seguintes mudanças para atender a demanda:

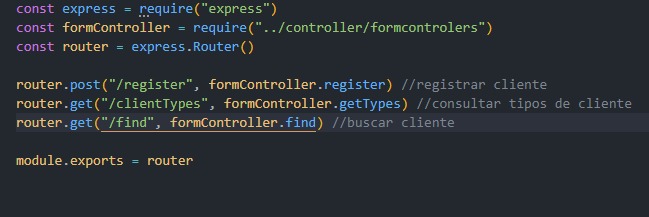


Figura 34 mudança do path para /find somente

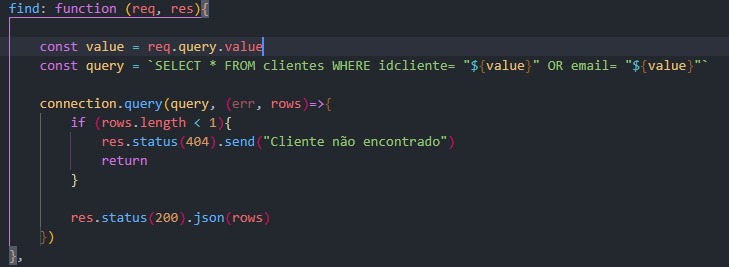


Figura 35 mudança na req para req.query.value

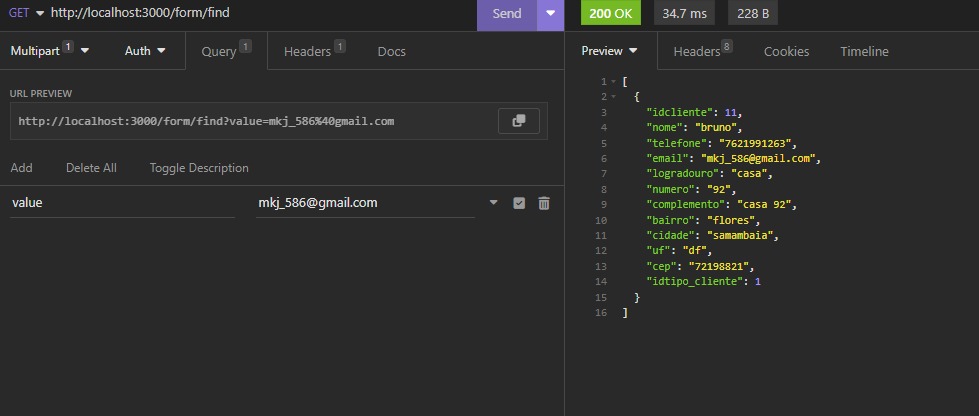


Figura 36 teste feito utilizando query params

**Atualização 27/03/2023**

Como query params não foi o requerido pelas regras da atividade, voltamos para a versão anterior na função e adicionamos mais um caminho como foi pedido na atividade.

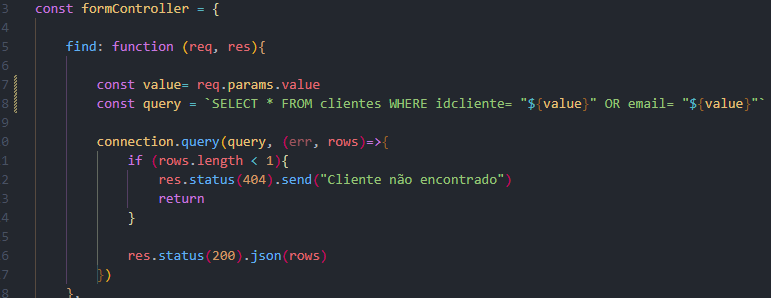


Figura voltando a const value para req.params.value

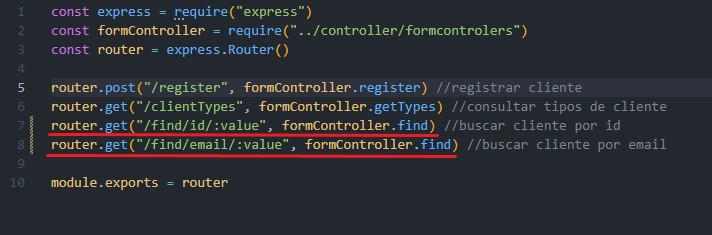


Figura adicionada novos rotas para atender as exigências da atividade.

4- Conclusão

Essa atividade foi de muito aprendizado, entender que as queries funcionam um pouco diferentes das que são passadas quando faz um script no Dbeaver, por exemplo. E principalmente de muito aprendizado no docker, dentre as várias pesquisas e tentativas que fiz, aprendi um pouco mais sobre volumes, criar um volume vazio e passar como option do banco de dados mysql, os dados inseridos sempre estarão no volume. Aprendi a criar uma imagem de um container com suas modificações com o comando “commit”, e ainda a como subir uma imagem para o dockerhub para que outras pessoas possam usá-las.

Dessa forma, e com esses conhecimentos adquiridos, me sinto mais capacitado a trabalhar com o docker quando for necessário.

Além disso, com a nova aula do dia 27/03/23, compreendi que devo estar mais atento aos detalhes, questionar mais quando uma atividade for passada, como também durante a atividade, quando surgir opções e que decisões devem ser tomados, sempre questionar ao tech leader qual dessas opções seguir para evitar problemas futuros.