Testes de Componentes

Função para criar a conta do restaurante:

A função abaixo serve para o restaurante fazer o cadastro de suas informações no sistema. Quando os dados são preenchidos no frontend, o backend recebe e trata esses dados, criando assim uma conta para aquele restaurante, e no final, ainda retorna para o restaurante uma mensagem dizendo se foi possível ou não criar a conta:

```
export async function createAccount(restaurantData) {
    try {
      const db = await openDB()
      await db.run(
        "INSERT INTO restaurantUserData (email, restaurant_name, cpf_cnpj, password, address, contact, time, image) VALUES (?,?,?,?,?,?,?,?)",
        [
          restaurantData.email,
          restaurantData.restaurantName,
          restaurantData.cpf_cnpj,
          restaurantData.password,
          '-',
          '-',
          '-',
          '1DBGw5tyRTCz538sQEBG2gB19d7BnOTCZ'
        ]
      )
      return {
        response: `${restaurantData.restaurantName} cadastrado com sucesso!!!`,
        status: true
      }
    } catch (err) {
      const error =  err.message.split(": ")[2].split(".")[1]
      return error
    }
  }
```

Fizemos a tratativa para caso a pessoa tente cadastrar um email ou cpf/cnpj já cadastrados (visto que isso é de uso único), fazemos a tratativa para voltar qual dos dois já existe, e pedir para inserir outro.

Para os testes, criamos uma função que vai receber um objeto com dados fictícios, e retornar se foi possível ou não criar esses dados. Repare na função externa com o nome “encrypt”. Essa função tem o objetivo de criptografar a senha e não salvar no banco como foi informada pelo usuário.

```
async function testCreateAccount(user, esp)
{
  user.password = encrypt(user.password)
  const enc = await createAccount(user)
  if(enc.status === esp)
  {
    return "Restaurante cadastrado com sucesso no banco de dados!!!"
  }
  else
  {
    return `Erro ao cadastrar novo restaurante. Erro: ${enc} já cadastrado!!!`
  }
}
```

Para os testes, criamos um objeto com os dados, e chamamos a função:

```
const user = {
  email: 'alphaTeste@gmail.com',
  restaurantName: 'Teste teste',
  cpf_cnpj: '98498245687',
  password: '@Teste1234'
}

console.log(await testCreateAccount(user, true))
```

A resposta que teremos é:

```
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src> node createRestaurante.js
Restaurante cadastrado com sucesso no banco de dados!!!
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src>
```

Note que ele foi cadastrado no banco de dados. Agora, e se rodarmos novamente o mesmo teste, o que deve acontecer?

```
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src> node createRestaurante.js
Erro ao cadastrar novo restaurante. Erro: cpf_cnpj já cadastrado!!!
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src>
```

Nos informa que aquele CPF/CNPJ já está cadastrado no sistema, impossibilitando a criação da conta

Rodando o fluxo com um CPF diferente, temos a mesma resposta, visto que o email permanece igual ao de um já cadastrado

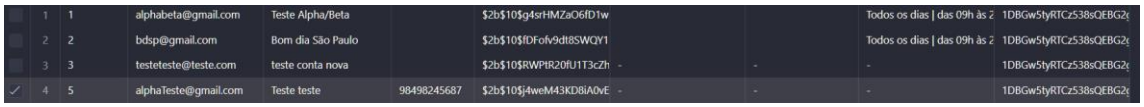
```
const user = {
  email: 'alphaTeste@gmail.com',
  restaurantName: 'Teste teste',
  cpf_cnpj: '123414454214',
  password: '@Teste1234'
}

console.log(await testCreateAccount(user, true))
```

Resposta que temos:

```
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src> node createRestaurante.js
Erro ao cadastrar novo restaurante. Erro: email já cadastrado!!!
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src>
```

E no banco, o restaurante foi cadastrado corretamente:



1	1	alphabeta@gmail.com	Teste Alpha/Beta		$2b$10$g4srHMZaO6fD1w			Todos os dias | das 09h às 2	1DBGw5tyRTCz538sQEBG2(
2	2	bdsp@gmail.com	Bom dia São Paulo		$2b$10$fDFofv9dt8SWQY1			Todos os dias | das 09h às 2	1DBGw5tyRTCz538sQEBG2(
3	3	testeteste@teste.com	teste conta nova		$2b$10$RWPtR20fU1T3cZh	-	-	-	1DBGw5tyRTCz538sQEBG2(
4	5	alphaTeste@gmail.com	Teste teste	98498245687	$2b$10$j4weM43KD8iA0vE	-	-	-	1DBGw5tyRTCz538sQEBG2(

Função para registrar um novo prato:

A função abaixo tem o objetivo de salvar um prato cadastrado pelo restaurante no banco de dados. Fazendo a tratativa dos dados informados, da imagem do prato e se o prato vai ou não aparecer para os clientes (Funcionalidade essa que o restaurante pode sempre escolher se vai mostrar ou não).

```
export async function addItemOnMenu(itemData)
{
    let activated = 0
    if(itemData.activated)
    {
        activated = 1
    }
    try {
        const db = await openDB()
        await db.run("INSERT INTO menuData (id_restaurant, dish_name, description, image, category, price, activated) VALUES (?,?,?,?,?,?,?)",
        [
            itemData.idRestaurant,
            itemData.dishName,
            itemData.description,
            itemData.imageURL,
            itemData.category,
            itemData.price,
            activated
        ])
        await closeDB()
        return true
    } catch (error) {
        console.log(error)
        return error
    }
}
```

Para os testes, criamos uma função que vai enviar um objeto para a função, para que ela possa salvar esse novo pedido:

```
async function testAddItemOnMenu(item, esp)
  {
    const enc = await addItemOnMenu(item)
    if(enc === esp)
    {
      return "Pedido cadastrado com sucesso no banco de dados!!!"
    }
    else
    {
      return `Erro ao cadastrar o pedido. Erro: ${enc}!!!`
    }
  }
```

Para o teste, criamos um objeto que recebe os dados fictícios, e chama a função:

```
const item = {
  idRestaurant: 1,
  dishName: 'Macarrão',
  description: 'Macarrão com Molho Verde',
  imageURL: '1DBGw5tyRTCz538sQEBG2gB19d7BnOTCZ',
  category: 'Prato',
  price: 12,
  activated: true,
}

console.log(await testAddItemOnMenu(item, true))
```

Note que a propriedade “imageURL” recebe um código. Esse código vem de outro função externa a essa que faz a tratativa da imagem selecionada pelo restaurante, e quando essa imagem é salva, retorna um ID para a imagem poder ser acessada, e é esse ID que salvamos no banco.

Rodando o teste, teremos a seguinte resposta:

```
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src> node addItemOnMenu.js
Pedido cadastrado com sucesso no banco de dados!!!
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src>
```

E olhando o banco, nós temos:

	* ID INTEGER	* id_restaurant INTEGER	* dish_name TEXT	* description TEXT	* image TEXT	* category TEXT	* price REAL	* activated INTEGER
1	1	1	Hamburguer	Hamburguer Topzão	1XEXgJ-McKy00XG0LgKIPYI	Prato	30.0	1
2	3	1	bolinho	Bolo de chocolate com cob	146z7ZwejJIQxlgt5tH640EG	Sobremesa	40.0	1
3	4	1	Bolo de chocolate com non	Bolo de chocolate com non	1Z6FyQvgr-FARaqod2YWev	Sobremesa	5000.0	1
4	12	1	Suco gostoso	Suco geladinho	1QagCeTC7cSk72YIgJbzFFZ	Bebida	15.0	1
5	23	1	Macarrão	Macarrão com Molho Verde	1DBGw5tyRTCz538sQEBG2(	Prato	12.0	1

Agora, caso uma propriedade venha vazia, o item não será cadastrado, e o seguinte erro será gerado:

```
[Error: SQLITE_CONSTRAINT: NOT NULL constraint failed: menuData.category] {
  errno: 19,
  code: 'SQLITE_CONSTRAINT'
}
Erro ao cadastrar o pedido. Erro: Error: SQLITE_CONSTRAINT: NOT NULL constraint failed: menuData.category!!!
PS C:\Users\JOÃO\Desktop\WaiterTech\E7-\src\waiter-tech-api\src>
```

Mas no frontend, é feito uma limitação completa para o restaurante só cadastrar o prato, caso todos os campos sejam preenchidos.