**Sistema Web oMenu**

O objetivo do projeto é desenvolver uma solução para que pessoas com deficiência visual tenham a possibilidade de realizar pedidos em restaurantes com maior autonomia, usando recursos sonoros do próprio sistema sem haver a necessidade de auxílio de qualquer outra ferramenta de acessibilidade.

Nosso sistema web é desenvolvido na Linguagem Python, com o Framework Flask e seguindo os princípios da arquitetura REST (Representational State Transfer).

**Bibliotecas utilizadas:**

asgiref==3.7.2

bcrypt==4.1.2

blinker==1.7.0

click==8.1.7

colorama==0.4.6

Flask==3.0.3

Flask-JWT-Extended==4.6.0

greenlet==3.0.3

itsdangerous==2.2.0

MarkupSafe==2.1.5

peewee==3.17.3

psycopg2==2.9.9

PyJWT==2.8.0

python-dotenv==1.0.1

sqlparse==0.4.4

typing\_extensions==4.11.0

tzdata==2023.3

Werkzeug==3.0.2

A instalação das bibliotecas deve ser feita na pasta:

Entrega Técnica – Carol/backend/

Digitando no console: pip install NOMEDABIBIOTECA

**Como rodar o projeto?**

Após a instalação das bibliotecas utilizadas, para iniciar a execução do backend, é recomendado rodar o comando:

“python main.py” no console, dentro da pasta: Entrega Técnica – Carol/backend/

**Organização das pastas, subpastas e arquivos**

Dentro da pasta principal (backend), temos as pastas: (bancodedados), (controladores), (funcoes), (rotas) e os arquivos: “.env”, “configuração.py”, “main.py”.

**.env**

Arquivo de configuração das variáveis de ambiente.

URL\_DATABASE: As credenciais de acesso ao banco de dados em nuvem ElephantSQL.

JWT\_SECRET\_KEY: Senha utilizada como chave para criptografia da configuração do JWT (JASON Web Token).

**configuracao**

Principal arquivo de configuração do sistema. Aqui estabelecemos as configurações necessárias para o funcionamento do sistema com os recursos já desenvolvidos.

**Bibliotecas (Métodos) utilizadas:**

os

dotenv (load\_dotenv)

flask\_jwt\_extended (JWTManager)

rotas.administrador (administrador\_rotas)

rotas.cliente (cliente\_rotas)

bancodedados.bancodedados (db)

**Detalhamento da execução**

“configure\_all” é a função que executa todas as configurações do sistema. Recebe como parâmetro a instância do Flask “app”.

“configure\_rotas” configuração das rotas na instância do Flask. Registra as Blueprints.

“configure\_db” configuração e conexão com o banco de dados.

“configure\_jwt” configuração que permite utilização de token de acesso as rotas protegidas do sistema.

O objetivo de criar um arquivo de configuração geral é deixar o código do programa principal “main.py” o mais limpo possível.

**main**

Arquivo principal. Quando executado, todo as rotas já implementadas ficam disponíveis.

**Bibliotecas (Métodos) utilizadas:**

flask (Flask)

configuracao (configure\_all)

**Detalhamento da execução**

Inicialmente criamos a instância do flask na variável (app).

Em seguida iniciamos o servidor com o método “.run”. Por padrão o servidor criado no flask executa na porta 5000, em nosso projeto a porta foi alterada para 3000.

**Banco de Dados do Sistema Web oMenu**

Foi desenvolvido um banco de dados relacional usando (PostgreSQL) e ElephantSQL para manter o banco em nuvem.

Também foi utilizado o ORN (Object Relacional Mapping) Peewee para fazer a interação com o banco de dados.

Na pasta (bancodedados), temos a subpasta (modelos) onde estão as classes criadas com os modelos compatíveis com as tabelas do banco, possibilitando o uso de um ORN para interagir com nossa base de dados.

Ainda na pasta (bancodedados) temos o arquivo “bancodedados.py”

**bancodedados**

**Bibliotecas (Métodos) utilizadas:**

os

pewee (PostgresqlDatabase)

dotenv (load\_dotenv)

**Detalhamento da execução**

Criamos a função (db) que será importada no arquivo de configuração geral da aplicação bem como nos modelos das classes.

Usando o método “PostgresqlDatabase” criamos uma conexão do tipo PostgreSQL. Usamos a biblioteca “os” com a função “getenv” para carregar as configurações das variáveis de ambiente, e passamos como parâmetro a variável de ambiente (DATABASE\_URL) que contêm as informações de conexão com o banco de dados em nuvem ElephantSQL.

**Modelos**

Na pasta (modelos), as classes foram divididas em arquivos separados. Assim cada tabela possui arquivo com seu nome.

**Categorias**

Tabela criada para agrupar os produtos por categoria.

**Bibliotecas (Métodos) utilizadas:**

peewee (Model, Charfield)

**Conexão importada**

bancodedados.bancodedados (db)

**Detalhamento da execução**

Criamos a classe do modelo base, que indica o banco de dados utilizado através da função “db” importada no arquivo.

Na sequência criamos a classe “Categorias” que utiliza o modelo base e possui as seguintes propriedades:

descricao: Campo de texto (string), não pode ser nulo.

**Mesas**

Tabela criada salvar o cadastro das mesas/clientes do sistema.

**Bibliotecas (Métodos) utilizadas:**

peewee (Model, Charfield)

**Conexão importada**

bancodedados.bancodedados (db)

**Detalhamento da execução**

Criamos a classe do modelo base, que indica o banco de dados utilizado através da função “db” importada no arquivo.

Na sequência criamos a classe “Mesas” que utiliza o modelo base e possui as seguintes propriedades:

nome: Campo de texto (string), não pode ser nulo e deve ser único.

senha: Campo de texto (string), não pode ser nulo.

**Pedidos**

Tabela criada para os pedidos realizados por mesas/clientes.

**Bibliotecas (Métodos) utilizadas:**

peewee (Model, Integerfield, TextField, ForeignKeyField)

**Modelos importados:**

bancodedados.modelos.Mesas (Mesas)

**Conexão importada:**

bancodedados.bancodedados (db)

**Detalhamento da execução**

Criamos a classe do modelo base, que indica o banco de dados utilizado através da função “db” importada no arquivo.

Na sequência criamos a classe “Pedidos” que utiliza o modelo base e possui as seguintes propriedades:

mesa\_id: Campo numérico (int), não pode ser nulo. Chave estrangeira, que faz referência a coluna (id) da tabela “Mesas”.

observacao: Campo de texto (string) maior que CharField.

valor: Campo numérico (int), não pode ser nulo.

**Produtos**

Tabela criada para salvar os produtos cadastrados no sistema.

**Bibliotecas (Métodos) utilizadas:**

peewee (Model, CharField, Integerfield, TextField, ForeignKeyField)

**Modelos importados:**

bancodedados.modelos.Categorias (Categorias)

**Conexão importada:**

bancodedados.bancodedados (db)

**Detalhamento da execução**

Criamos a classe do modelo base, que indica o banco de dados utilizado através da função “db” importada no arquivo.

Na sequência criamos a classe “Pedidos” que utiliza o modelo base e possui as seguintes propriedades:

mesa\_id: Campo numérico (int), não pode ser nulo. Chave estrangeira, que faz referência a coluna (id) da tabela “Mesas”;

observacao: Campo de texto (string) maior que CharField;

valor: Campo numérico (int), não pode ser nulo;

categoria\_id: Campo numérico (int), chave estrangeira que faz referência a coluna (id), da tabela “Categorias”;

imagem: Campo de texto (string), será salvo o endereço da imagem no servidor de mídea;

video: Campo de texto (string), será salvo o endereço do vídeo no servidor de mídia.

**Usuários**

Tabela criada salvar o cadastro dos usuários do sistema.

**Bibliotecas (Métodos) utilizadas:**

peewee (Model, Charfield)

**Conexão importada**

bancodedados.bancodedados (db)

**Detalhamento da execução**

Criamos a classe do modelo base, que indica o banco de dados utilizado através da função “db” importada no arquivo.

Na sequência criamos a classe “Usuarios” que utiliza o modelo base e possui as seguintes propriedades:

username: Campo de texto (string), não pode ser nulo e deve ser único.

senha: Campo de texto (string), não pode ser nulo.

**funcoes**

Onde ficarão todas as funções desenvolvidas para processamento de alguma informação do sistema Ex: (Calculo de total, conferencia de produtos cadastrados). No momento as únicas funções na pasta fazem parte da execução da rota (INCOMPLETA) de cadastro de pedidos. Ao finalizar a rota a descrição será atualizada.

**Rotas do Sistema Web oMenu**

Na pasta de rotas temos dois arquivos principais onde estão separadas as rotas de Administrador e rotas de Cliente (administrador.py e cliente.py).

**Administrador**

**Bibliotecas (Métodos) utilizadas:**

flask (Blueprint)

flsak\_jwt\_extended (jwt\_required)

**Detalhamento da execução**

Primeiro é realizada a importação de todos os controladores de Administrador.

Na sequência criamos a Blueprint (administrador\_rotas), conjunto de rotas que será executado na aplicação (\_\_name\_\_). Essa Blueprint será importada no arquivo de configuração geral da aplicação (configuracao.py)

Por fim, temos a criação das rotas usando decorators para associar as funções que serão executadas a URL passada como parâmetro em (.route). As funções executadas retornam outras funções, os controladores.

A rota de login é a única sem proteção de token.

**Cliente**

**Bibliotecas (Métodos) utilizadas:**

flask (Blueprint)

flsak\_jwt\_extended (jwt\_required)

**Detalhamento da execução**

Primeiro é realizada a importação de todos os controladores de Cliente.

Na sequência criamos a Blueprint (cliente\_rotas), conjunto de rotas que será executado na aplicação (\_\_name\_\_). Essa Blueprint será importada no arquivo de configuração geral da aplicação (configuracao.py)

Por fim, temos a criação das rotas usando decorators para associar as funções que serão executadas a URL passada como parâmetro em (.route). As funções executadas retornam outras funções, os controladores.

A rota de login é a única sem proteção de token.

**Controladores do Sistema Web oMenu**

Os controladores foram testadas utilizando o Insomnia para simular o frontend.

Todas as rotas protegidas por token exigem login prévio.

Após logar no sistema, é necessário copiar o token de acesso gerado e colar em AUTH/Bearer Token

**Cliente (MESAS)**

**Login do cliente/mesa**

**POST /login**

http://127.0.0.1:3000/loginCliente/mesa01&abcd123

Controlador que permite o usuário cadastrado realizar o login no sistema.

**Bibliotecas (Métodos) utilizadas:**

flask (make\_response, jsoninfy)

flask\_jwt\_extended (create\_access\_token)

bcrypt

**Tabelas Banco de Dados utilizadas:**

Mesas

**Detalhamento da execução**

O controlador recebe como parâmetro as informações de login do cliente (mesa\_nome e mesa\_senha).

Em seguida é feita a validação, através de uma busca na tabela “Mesas” que retorna a mesa cadastrada com o mesmo nome passado como parâmetro (mesa\_nome). Caso não haja retorno, o sistema retorna mensagem de erro e código de status.

Logo após, será conferida se as senhas conferem. Usando a biblioteca “bcrypt” e o método “checkpw” para conferir se a senha passada como parâmetro (mesa\_senha) é igual a senha cadastrada no banco. Não sendo a mesma senha, o sistema retorna mensagem de erro e código de status.

Com as informações validadas, utilizamos o método “create\_access\_token” para gerar um token de validação no sistema. Esse token contém como informação principal de identidade o “nome” da mesa/usuário cadastrado. Não conseguindo gerar o token, o sistema retorna mensagem de erro e código de status.

Por fim, será retornado o token do usuário/mesa logado no sistema.

**Listagem de todos os produtos cadastrados no banco (Cardápio)**

**GET /produtos**

**Rota protegida por token**

http://127.0.0.1:3000/cardapio/

Essa é a rota que será chamada quando o usuário logado. Deverá listar todos os produtos cadastrados.

**Bibliotecas (Métodos) utilizadas:**

flask (make\_response, jsoninfy)

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Mesas

Produtos

**Detalhamento da execução**

O controlador inicia, guardando na variável (mesa) a identidade armazenada no token, através do método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Mesas” que retorna a mesa cadastrada com o mesmo nome registrado como identidade no token. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com a identidade do usuário/mesa validada, será armazenado em “produtos” em formato de dicionário os produtos cadastrados no banco. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com as informações validadas, criamos a resposta que será uma lista de objetos contendo as propriedades dos produtos cadastrados.

OBS: A ideia é acrescentar no frontend a funcionalidade de leitura do nome e preço dos produtos (no modo de acessibilidade). Em uma espécie de carrossel, os produtos serão exibidos um por um com descrição sonora das propriedades acima citadas. Ainda não sabemos qual biblioteca usar. Provavelmente o desenvolvimento da parte visual será utilizando TypeScript e React.

**Detalhar Produto**

**GET /produtos/:id**

**Rota protegida por token**

http://127.0.0.1:3000/produto/2

Controlador que permite o usuário logado obter informações detalhadas de um dos produtos cadastrados.

**Bibliotecas (Métodos) utilizadas:**

flask (make\_response, jsoninfy);

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Mesas

Produtos

**Detalhamento da execução**

O controlador recebe como parâmetro o id do produto a ser detalhado (produto\_id). Também recebe o token do usuário/mesa que realiza a solicitação, salvando na variável (mesa) a identidade armazenada no token, através do método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Mesas” que retorna a mesa cadastrada com o mesmo nome registrado como identidade no token. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Após validar a mesa, será realizada mais uma busca no banco, desta vez na tabela “Produtos”, para validar a existência do produto. A busca deverá retornar o produto cujo id é igual ao (produto\_id) passado como parâmetro. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com a identidade do usuário/mesa validada, e o produto também validado, será criada a resposta que retornará uma lista com o dicionário do produto encontrado.

OBS: Os recursos de acessibilidade sonora da aplicação serão implementados no frontend.

**Realizar Pedido**

**POST /pedidos**

**Rota protegida por token**

Rota em desenvolvimento.

**Administrador**

**POST /login**

<http://127.0.0.1:3000/login>

**Exemplo de entrada:**

**{**

**"username":"albert",**

**"senha":"1234"**

**}**

Controlador que permite o usuário cadastrado realizar o login no sistema.

**Bibliotecas (Métodos) e funções utilizadas:**

flask (request, make\_response, jsoninfy);

flask\_jwt\_extended (create\_access\_token)

bcrypt

conferir\_username\_senha (Função)

**Tabelas Banco de Dados utilizadas:**

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no corpo da requisição (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Logo após, será conferida se as senhas conferem. Usando a biblioteca “bcrypt” e o método “checkpw” para conferir se a senha passada como parâmetro (senha) é igual a senha cadastrada no banco. Não sendo a mesma senha, o sistema retorna mensagem de erro e código de status.

Com as informações validadas, utilizamos o método “create\_access\_token” para gerar um token de validação no sistema. Esse token contém como informação principal de identidade o “username” do usuário cadastrado. Não conseguindo gerar o token, o sistema retorna mensagem de erro e código de status.

Por fim, será retornado o token do usuário logado no sistema.

**Cadastrar Produto**

**POST /produtos**

**Rota protegida por token**

<http://127.0.0.1:3000/produto/>

**Exemplo de entrada:**

{

"nome":"Pizza BriePimenta",

"detalhamento":"Pizza com recheio de queijo brie e geleia de pimenta",

"valor":1500,

"categoria\_id":4,

"imagem":"www.imagem.com",

"video":"www.video.com"

}

Controlador que permite o usuário logado cadastrar um novo produto no sistema.

**Bibliotecas (Métodos) utilizadas:**

flask (request, make\_response, jsoninfy)

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Categorias

Produtos

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição. Também é armazenado na variável (usuario) a identidade do usuário presente no token, usando o método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no token do usuário (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Após a validação das credenciais do usuário, é feita a validação da categoria do produto a ser cadastrado, conferindo se o id de categoria passado no corpo da requisição é válido (Se existe uma categoria cadastrada no banco com o mesmo id). É realizada uma busca na tabela “Categorias”, para encontrar a categoria com id igual ao (categoria\_id) passado no corpo da requisição. Caso não haja nenhuma categoria cadastrada com id igual ao id de busca, será retornada uma mensagem de erro e o código de status.

A última validação será feita buscando por produto cadastrado no banco de dados com o mesmo nome que foi passado no corpo da requisição (req.nome). Fazemos uma busca na tabela produtos, buscando por produto cadastrado com mesmo nome passado na requisição. Com retorno positivo, ou seja, caso exista um produto de mesmo nome já cadastrado, será retornada uma mensagem de erro e o código de status.

Com todas as validações concluídas, armazenamos em uma variável (produto\_cadastrado) o retorno da ação de criar um novo registro na tabela “Produtos”. Cada propriedade passada na requisição, será salva como propriedade do novo produto. Caso não exista retorno, o sistema envia mensagem de erro e código de status.

Finalizando, o sistema retorna em formato JSON, uma mensagem de sucesso e o código de status.

**OBS:** O módulo de upload de arquivos será implementado nessa rota. Será utilizado a Biblioteca “boto3” (AWS – SDK). Também será alterada a forma de recepção dos dados da requisição para multipart/form-data, permitindo o usuário enviar arquivo de imagem e vídeo.

**Cadastrar Cliente/Mesa**

**POST /mesa**

**Rota protegida por token**

<http://127.0.0.1:3000/mesa/>

**Exemplo de entrada:**

{

"nome":"mesa06",

"senha":"abcd123"

}

Controlador que permite usuário logado cadastrar um novo cliente/mesa no sistema.

**Bibliotecas (Métodos) utilizadas:**

flask (request, make\_response, jsoninfy)

bcrypt

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Categorias

Mesas

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição. Também é armazenado na variável (usuario) a identidade do usuário presente no token, usando o método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no token de acesso (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

A segunda validação é feita através de uma busca na tabela Mesas, com o objetivo de conferir se existe registro na tabela “Mesas” com nome igual ao nome passado no corpo da requisição (req[‘nome’]). Caso tenha registro com mesmo nome, o sistema retorna mensagem de erro e código de status.

Após a validação das credenciais do usuário e garantida a disponibilidade do nome, será criada a senha criptografada para salvar no banco. Criamos uma variável (senha\_hash) onde será armazenada a senha gerada usando a biblioteca “bcrypt” e seu método “hashpw” (O método “hashpw, recebe como parâmetro a senha em formato original (req[‘senha’]) junto com seu formato de codificação (UTF-8), mais o método “gensalt” que gera uma string aleatória para a senha fornecida na requisição. Caso ocorra erro no processo, será retornada uma mensagem de erro e o código de status.

Com as validações concluídas e a (senha\_hash) gerada, armazenamos em uma variável (mesa\_cadastrada) o retorno da ação de criar um novo registro na tabela “Mesas”. A propriedade nome do novo registro será idêntica ao passado na requisição (req[‘nome’]) e a senha será a (senha\_hash) gerada pelo sistema.

Finalizando, o sistema retorna em formato JSON, uma mensagem de sucesso e o código de status.

**Listar Pedidos**

**GET /pedido**

**Rota protegida por token**

Rota em desenvolvimento

**Atualizar Produto Cadastrado**

**PUT /produtos/**

**Rota protegida por token**

<http://127.0.0.1:3000/produto/>

**Exemplo de entrada:**

{

"produto\_id": 4,

"descricao":"Piza Camarão Com Chocolate",

"valor": 90000,

"categoria\_id": 4,

"audio":"www.audio.com",

"imagem":"www.imagem.com",

"video":"www.video.com"

}

Controlador que permite o usuário logado atualizar as informações de um produto cadastrado.

**Bibliotecas (Métodos) utilizadas:**

flask (request, make\_response, jsoninfy);

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Categorias

Produtos

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição. Também é armazenado na variável (usuario) a identidade do usuário presente no token, usando o método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no token de acesso (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

A segunda validação é feita através de uma busca na tabela “Produtos”, com o objetivo de conferir se existe registro na tabela “Produtos” com id igual ao (req[‘produto\_id’]) passado no corpo da requisição. Caso não tenha registro com mesmo id, o sistema retorna mensagem de erro e código de status.

Após a validação das credenciais do usuário e do produto, será feita a última validação no id de categoria passado no corpo da requisição (req[‘categoria\_id’]), buscando na tabela “Categorias” um registro com mesmo id. Caso não exista categoria cadastrada com o id correspondente, será retornada uma mensagem de erro e o código de status.

Com as validações concluídas utilizando o produto encontrado na validação de produto, fazemos atualização de todas as propriedades do produto, com os dados passados na requisição. A atualização encerra com o comando “.save()”, salvando as alterações no registro.

Finalizando, o sistema retorna em formato JSON, uma mensagem de sucesso e o código de status.

OBS: Na página de atualizar produto (frontend) a ideia é iniciar com o preenchimento de todos os campos de propriedades do produto em questão e permitir a alteração de todas as propriedades convenientes. Assim a rota espera receber informação correspondente de todas as propriedades que permitem mudança. Qualquer alteração será enviada com novo valor e, propriedades que não forem alteradas recebem (novamente) a mesma informação.

**Excluir Produto Cadastrado**

**DELETE /produtos/**

**Rota protegida por token**

<http://127.0.0.1:3000/produto/>

**Exemplo de entrada:**

{

"produto\_id": 10

}

Controlador que será chamado quando o usuário logado quiser excluir um dos produtos cadastrados.

**Bibliotecas (Métodos) utilizadas:**

flask (request, make\_response, jsoninfy)

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Produtos

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição. Também é armazenado na variável (usuario) a identidade do usuário presente no token, usando o método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no token de acesso (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

A segunda validação é feita através de uma busca na tabela “Produtos”, com o objetivo de conferir se existe registro na tabela “Produtos” com id igual ao (req[‘produto\_id’]) passado no corpo da requisição. Caso não tenha registro com mesmo id, o sistema retorna mensagem de erro e código de status.

Com as validações concluídas, será registrado na variável (produto\_deletado) o retorno da ação de exclusão do registro cujo id é igual ao (req[‘produto\_id’]) passado no corpo da requisição. Caso não haja retorno à variável (produto\_deletado), o sistema retorna mensagem de erro e código de status.

Finalizando, o sistema retorna em formato JSON, uma mensagem de sucesso e o código de status.

**Listagem de todos os produtos cadastrados no banco**

**GET /produtos**

**Rota protegida por token**

<http://127.0.0.1:3000/produtos/>

Controlador chamado quando o usuário logado desejar visualizar todos os produtos cadastrados (cardápio). Deverá listar todos os produtos cadastrados.

**Bibliotecas (Métodos) utilizadas:**

flask (make\_response, jsoninfy)

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Produtos

Usuarios

**Detalhamento da execução**

O controlador inicia, guardando na variável (usuario) a identidade armazenada no token, através do método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna o usuário cadastrado com o mesmo nome registrado como identidade no token. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com a identidade do usuário validada, será armazenado em “produtos” em formato de dicionário os produtos cadastrados no banco. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com as informações validadas, criamos a resposta que será uma lista de objetos contendo as propriedades dos produtos cadastrados.

**Detalhar Produto**

**GET /produtos/:id**

**Rota protegida por token**

<http://127.0.0.1:3000/produto/2>

Controlador que permite o usuário logado obter informações detalhadas de um dos produtos cadastrados.

**Bibliotecas (Métodos) utilizadas:**

flask (make\_response, jsoninfy)

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Produtos

Usuarios

**Detalhamento da execução**

O controlador recebe como parâmetro o id do produto a ser detalhado (produto\_id). Também recebe o token do usuário que realiza a solicitação, salvando na variável (usuario) a identidade armazenada no token, através do método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna o usuario cadastrado com o mesmo nome registrado como identidade no token. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Após validar ao usuário, será realizada mais uma busca no banco, desta vez na tabela “Produtos”, para validar a existência do produto. A busca deverá retornar o produto cujo id é igual ao (produto\_id) passado como parâmetro. Caso não haja retorno, o sistema envia mensagem de erro e código de status.

Com a identidade do usuário validada, e o produto também validado, será criada a resposta que retornará uma lista com o dicionário do produto encontrado.

**Cadastrar Usuário**

**POST /usuário**

**Rota protegida por token**

<http://127.0.0.1:3000/usuario>

**Exemplo de entrada:**

{

"username":"graça",

"senha":"1234"

}

Controlador que permite usuário logado cadastrar um novo usuário no sistema.

**Bibliotecas (Métodos) utilizadas:**

flask (request, make\_response, jsoninfy)

bcrypt

flask\_jwt\_extended (get\_jwt\_identity)

**Tabelas Banco de Dados utilizadas:**

Categorias

Mesas

Usuarios

**Detalhamento da execução**

O controlador armazena na variável (req) os dados do corpo da requisição. Também é armazenado na variável (usuario) a identidade do usuário presente no token, usando o método “get\_jwt\_identity”.

Em seguida é feita a validação, através de uma busca na tabela “Usuarios” que retorna usuário cadastrado com o mesmo nome de usuário passado no token de acesso (username). Caso não haja retorno, o sistema envia mensagem de erro e código de status.

A segunda validação é feita através de outra busca na tabela “Usuarios”, com o objetivo de conferir se existe registro na tabela “Usuarios” com nome de usuário igual ao nome passado no corpo da requisição (req[‘username’]). Caso tenha registro com mesmo nome de usuário, o sistema retorna mensagem de erro e código de status.

Após a validação das credenciais do usuário e garantida a disponibilidade do nome de usuário, será criada a senha criptografada para salvar no banco. Criamos uma variável (senha\_hash) onde será armazenada a senha gerada usando a biblioteca “bcrypt” e seu método “hashpw” (O método “hashpw, recebe como parâmetro a senha em formato original (req[‘senha’]) junto com seu formato de codificação (UTF-8), mais o método “gensalt” que gera uma string aleatória para a senha fornecida na requisição. Caso ocorra erro no processo, será retornada uma mensagem de erro e o código de status.

Com as validações concluídas e a (senha\_hash) gerada, armazenamos em uma variável (usuario\_cadastrado) o retorno da ação de criar um novo registro na tabela “Usuarios”. A propriedade username do novo registro será idêntica ao passado na requisição (req[‘username’]) e a senha será a (senha\_hash) gerada pelo sistema.

Finalizando, o sistema retorna em formato JSON, uma mensagem de sucesso e o código de status.