Documentação do software

Tudo sobre o funcionamento do beesiness.

Criado por: Pedro Henrique Silva Quixabeira

Em: 05/09/2024

# Ideia

É ser um manual sobre algumas tecnologias e lógicas usadas no beesiness, suas sobre as funções, arquitetura e funcionamento como um todo. Com foco no auxílio dos desenvolvedores do Beesiness ou o mais fácil entendimento de quem quiser entender a lógica por trás de tudo.

Com o tempo irei adicionar mais coisas.

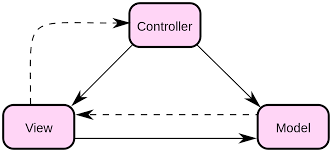
# Arquitetura

A Arquitetura que está sendo utilizada é a de camadas misturada com micros serviços. As camadas tem 3 níveis, a de apresentação, aplicação e dados. No código fonte do beesiness (source-code ou src em inglês) essas camadas estão divididas em 3 pastas:

Controller: Responsável por fazer o controle das requisições enviadas pelo front através das APIs, que executam micro serviços que fazem as verificações e requisições no banco de dados. A partir das respostas desses micros serviços, a API vai executar ou não o que o front pediu, sempre retornando algum código de status de resposta HTTP, código 200(tudo certo), 400(erro), 502(outro tipo de erro) sempre retornando um .json com alguma resposta, boa ou ruim.

Model: Onde todos os micros serviços estão, das verificações e conexão com o banco de dados.

Views: Tudo relacionado ao front, a camada de apresentação, que interage diretamente com o usuário, o html, css e o Javascript estão por lá.



# APIs

São como garçons em um restaurante, recebem um pedido de um cliente, e levam para a cozinha para eles tomarem as ações necessárias a partir disso, e claro, dependendo do pedido, o próprio garçom pode tomar as ações sem precisar de ir até a cozinha.

A diferença deles para as APIs, é que as APIs trabalham com métodos de requisição HTTP no lugar pedidos, a ideia é mesma, porém com algumas diferenças.

Para simplificar eu vou explicar a mesma coisa de maneiras diferentes, através de analogias.

GET PT-BR: O Cliente vai fazer um pedido para o garçom esperando que ele retorne algo em especifico, tipo pedir uma pizza de frango, você espera que o garçom vá e depois volte com uma pizza, de frango.

GET PC: O Front vai fazer uma requisição com o método GET para a API indexada a uma URL que em seguida vai devolver algo já pré-determinado no formato .json e algum código de status de resposta HTTP.

HEAD PC: Igual o GET porém nenhum dado em .json é retornado, somente o código de status de resposta HTTP.

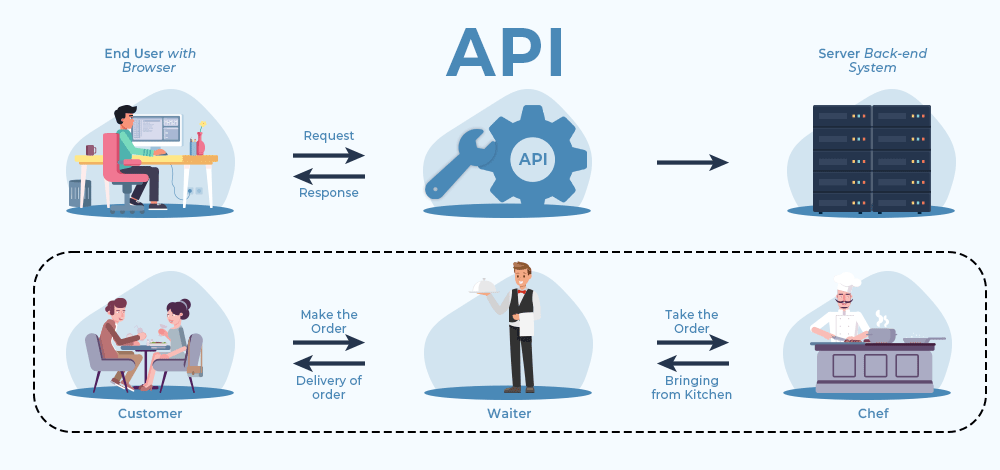
POST PT-BR: O cliente vai dar para o garçom um papel com os ingredientes de algo, o garçom vai pegar e vai dar para a cozinha para que eles façam a receita com os ingredientes dados. Depois de tudo, a cozinha vai retornar o pedido com a comida e o garçom vai dar a comida para o cliente.

POST PC: O front vai fazer uma requisição com o método POST enviando dados em um arquivo .json para a API. A API vai ler, mandar os dados para micro serviços(funções) que farão as verificações e demais tarefas que podem ser necessárias, se tudo der certo, vai devolver algo para o front, pode ser uma mensagem de “logado com sucesso”, “cadastrado com sucesso”, erros, enfim, qualquer coisa, porém, sempre em formato .json e enviando também algum código de status de resposta HTTP.

DELETE PT-BR: O Cliente vai pedir para o garçom jogar algo fora do estoque do restaurante.

DELETE PC: O Front vai fazer uma requisição com o método DELETE para a API indexada, enviando um .json com o algum dado que logicamente contextualize com o que vai ser deletado, como um id de um usuário por exemplo. Depois isso, os micros serviços vão executar a tarefa e depois a API vai retornar o resultado da requisição através de algum código de status de resposta HTTP.

PUT PC: O Front vai fazer uma requisição com o método PUT para a API indexada, enviando um .json com o algum dado que logicamente contextualize com o que vai ser atualizado, como um valor d um produto por exemplo. Depois isso, os micros serviços vão executar a tarefa e depois a API vai retornar o resultado da requisição através de algum código de status de resposta HTTP.



# Controller

## API – user\_login.py

Endereço: src\controller\API\login\user\_login.py

URL: api/user\_login

Responsável por receber as requisições através do método post e devolver a resultado da validação de um login de um usuário.

Formato do json de request:

{

    "email\_cpf": " ",

    "senha": " ”

}

O json será armazenado em uma variável:

login\_data = request.get\_json()

Os valores serão separados em duas variáveis:

email\_cpf = login\_data.get('email\_cpf')

password = login\_data.get('senha')

A partir do CPF ou email, será feita a pesquisa no banco de dados, caso haja uma conta com esses valores, serão retornados os dados ligados aos mesmos:

Id – email – cpf – senha(encriptada)

Se o CPF ou email não for encontrado, o erro será retornado pela API:

return jsonify({'login': False, 'error': 'Email ou CPF não encontrado'}), 400

Do contrário, a senha que o usuário enviou será encriptada e comparada com a senha já encriptada que veio do banco de dados, caso dê o mesmo resultado, o login foi efetuado com sucesso:

return jsonify({'login':True}), 200

Se não:

return jsonify({'login':False, 'error': 'Senha incorreta'}), 400

## API – new\_user\_account.py

Endereço: src\controller\API\new\_account\new\_user\_account.py

URL: api/create\_user\_account

Responsável por receber as requisições através do método post e devolver a resultado da validação da criação de uma conta de usuário.

Formato do json request:

{

    "nome": " ",

    "cpf": " ",

    "email": " ",

    "senha": " ",

    "data\_de\_nascimento": "xx/xx/xxxx"

}

O json será armazenado em uma variável:

create\_data = request.get\_json()

Os valores serão separados em cinco variáveis:

nome = create\_data.get('nome')

cpf = create\_data.get('cpf')

email = create\_data.get('email')

senha = create\_data.get('senha')

data\_nascimento = create\_data.get('data\_de\_nascimento')

Uma função de verificação será chamada e vai verificar, o CPF(Seja já está sendo utilizado), CNPJ(Se já está sendo utilizado) e a senha(se tem no mínimo 8 dígitos, uma letra maiúscula, uma minúscula, um número e um digito especial) e vai retornar o resultado em três parâmetros:

verified, errors, errors classes = verify all(cpf, email, senha)

“verified” vai retornar “True” ou “False”, se for “True”, todos os valores são válidos e podem continuar para a criação da conta do usuário, se for “False”, contém algum erro.

“errors" vai retornar os erros em uma list, podendo ser:

"CPF inválido." ; "Email inválido." ; "E-mail já cadastrado."

- Pode retornar mais de uma mensagem de erro.

- Se não houver erros, vai retornar “None”.

“errors classes” retorna a classe do erro, podendo ser:

“cpf”; “email”; “cnpj”

- Pode retornar mais de uma mensagem de erro.

- Se não houver erros, vai retornar “None”.

Se “verified” for “True”, não há erros, logo, dados do registro são válidos. A senha inserida pelo usuário será encriptada (1), os dados serão registrados no banco de dados(2) e uma resposta será dada ao front(3).

1 - hashed\_password = generate\_password\_hash(senha)

2 - db\_create\_user(nome, cpf, email, hashed\_password, data\_nascimento)

3 - return jsonify({"register": True}), 200

Se “verified” for “false”, os erros serão retornados para o front e o registro não irá acontecer.

return jsonify({"count\_error":len(errors), "register": False, "error":errors, "class":errors\_classes}), 400

- "count\_error":len(errors) é a quantidade de erros que foram captadas pelas verificações.

## API – new\_company\_account.py