Sugestão de melhoria

Proposta de melhoria: refatoração e escalabilidade

pasta *open-erp/app/models/services/bling***)**

Unificar métodos comuns

Todos os arquivos dentro da pasta: *find_order*, *find_orders*, *order*, *product*, *stock* e *update_order_status* possuem o mesmo código sendo repetido nas partes que envolvem requisições HTTP e também o token. Dessa forma, poderíamos ter um arquivo mais genérico que serviria de base, o qual pode ser herdado pelos outros arquivos, a fim de evitar essa duplicação de código.

Abaixo, um simples exemplo para ilustrar melhor o que poderia ser feito:

```
/*Classe que serviria de base*/
module Services
  module Bling
    class BlingService < ApplicationService

    (...)

        def bling_token
            @bling_token ||= BlingDatum.find_by(account_id: @
        end

        def authorization_headers
          {
            'Accept' => 'application/json',
            'Authorization' => "Bearer #{bling_token}"
          }
        end
```

```
        /*case - when das requisições. Isso poderia estar den
            /*que receberia o método HTTP e a url como parâme
        case method.to_s.downcase
                        when 'get' then Net::HTTP::Get
                        when 'post' then Net::HTTP::Post
                        when 'patch' then Net::HTTP::Patch
                        (...)
```

```
/*Arquivo que herdaria a classe genérica*/

module Services
    module Bling
      class FindOrders < BlingService

      (...)

      def find_order
          url = "https://www.bling.com.br(...)"
          response = make_request(method: 'get', url: url)
          response
        end

     (...)
```

Além disso, uma outra coisa que também poderia ser mais unificada a fim de evitar duplicação de código seria a centralização do lançamento de exceções.

Desacoplamento das classes com o padrão de design Strategy

Essa parte se relaciona um pouco com o que já foi proposto acima, pois o padrão Strategy permite que diferentes comportamentos ou comandos, como *find_order, find_orders, find_products, etc.*, sejam encapsulados em classes separadas, o que os tornam intercambiáveis.

Ao usar o padrão Strategy, cada comando é isolado em uma classe específica, como *FindOrderCommand* ou *FindOrdersCommand*. A classe principal do

serviço (BlingService) delega a execução para essas classes de comando. Isso melhora a organização do código, facilita a adição de novos comandos no futuro, e torna a manutenção mais simples, já que cada mudança pode ser feita de forma isolada.

Além disso, essa abordagem permite que o código seja mais modular, o que facilita a realização de testes unitários. Cada comando pode ser testado de forma independente, garantindo que o sistema funcione de forma correta ao adicionar ou modificar funcionalidades.

Abaixo, um simples exemplo para ilustrar melhor o que poderia ser feito:

```
/* Interface para os comandos */
module BlingCommand
  def execute
    raise NotImplementedError, '(...)'
  end
end
```

```
/* Implementação do comando find_order, implementando */
/*a interface do BlingCommand*/
class FindOrderCommand
  include BlingCommand

  def initialize(id, tenant)
    @id = id
    @tenant = tenant
  end

  def execute
    /*Lógica para encontrar um pedido específico*/
    (...)
  end
end
```

```
/*A classe BlingService vai usar o padrão Strategy*/
```

```
/*dessa forma, a execucação do comando é delegada */
/*para a estratégia correta*/

class BlingService
  def initialize(command)
    @command = command
  end

  def call
    @command.execute
  end
end
```

```
/*criação de uma instância da classe de comando e passa*/
/*para o BlingService*/
command = FindOrderCommand.new(id, tenant)
service = BlingService.new(command)
service.call

# Para encontrar vários pedidos
command = FindOrdersCommand.new(ids, tenant)
service = BlingService.new(command)
service.call
```

Usar Objetos de Valor

Quando métodos de serviços recebem múltiplos parâmetros, como *tenant, order_ids, e options*, o código pode se tornar difícil de manter e propenso a erros. Passar muitos parâmetros diretamente para métodos pode complicar o código e tornar ele mais difícil de entender e validar. Uma forma de simplificar e tornar o código mais robusto é usar **Objetos de Valor (Value Objects).**

Basicamente, os objetos de valor são classes que encapsulam um conjunto de dados relacionados. Por exemplo, em vez de passar um conjunto de parâmetros individuais, você cria um objeto que agrupa esses parâmetros em uma estrutura.

```
/*Objeto de valor que encapsularia os order_ids*/

class OrderRequest
  attr_reader :tenant, :order_ids, :options

  def initialize(tenant:, order_ids:, options: {})
    @tenant = tenant
    @order_ids = order_ids
    @options = options
  end

(...) /*Mais coisas, caso necessario*/
end
```

```
/*Em vez de passar os parâmetros individuais diretamente*/
/*para o método de serviço,*/
/*passamos o objeto OrderRequest.*/

module Services
  module Bling
class FindOrders
  def initialize(order_request)
    @order_request = order_request
  end

  (...)
```

```
/*instância de OrderRequest sendo passada para o serviço*/

order_request = OrderRequest.new(
  tenant: 'tenant_id',
  order_ids: [123, 456, 789],
  options: { some_option: 'value' }
)
```

```
service = FindOrders.new(order_request)
service.call
```

pasta *open-erp/blob/main/app/controllers/api/v1***)**

Estratégia de cache para melhoria de performance

Quando uma requisição é feita, a aplicação verifica primeiro se os dados necessários já estão no cache:

- **Se os dados estiverem no cache:** Eles são retornados imediatamente, sem consultar o banco de dados.
- **Se os dados não estiverem no cache:** A aplicação consulta o banco de dados, armazena os dados no cache, e os retorna para a requisição.

Ou seja, dessa forma conseguimos evitar a sobrecarga de consultas desnecessárias ao Banco de Dados, já que é algo que corre o risco de acontecer no momento nos arquivos de controllers.

Exemplo prático para ficar mais claro:

```
if cached_product
    render json: JSON.parse(cached_product)
else
    @product = Product.find(params[:id]) /* BD*/
```

Uso de microsserviços para melhorar a escalabilidade

Alguns arquivos dentro dessa pasta, como: products_controller.rb e orders_controller.rb são mais simples e diretos, mas outros como purchase_order_controller.rb já é um pouco mais robusto. Algo que poderia ser feito é aplicar o conceito de microserviço. Isso porque os microserviços permitem que você separe as responsabilidades, facilitando a escalabilidade, o desenvolvimento, e a manutenção. Por enquanto, o projeto se resume mais ao CRUD de produtos, mas caso no futuro isso venha a ficar mais complexo

dentro desses arquivos que já existem ou de novos que precisem ser criados, um microserviço focado em inventário seria uma boa escolha:

- **Serviço de Inventário:** Um serviço separado que lida com todas as operações de inventário, como adicionar produtos e atualizar quantidades.

```
class InventoryController < ApplicationController
  def add
    product_id = params[:product_id]
    quantity = params[:quantity]
    store_entrance = params[:store_entrance]
    account_id = params[:account_id]

    /*Lógica de adicionar inventário aqui*/

    render json: { status: 'success', message: 'Inventory add
  end

  def update
    product_id = params[:product_id]
    new_quantity = params[:new_quantity]

    /*Lógica de atualização de inventário aqui*/

    render json: { status: 'success', message: 'Inventory upd
  end
end
```

Aí, usamos esse serviço dentro do arquivo que queremos:

```
# app/controllers/api/v1/purchase_products_controller.rb
module Api
  module V1
    class PurchaseProductsController < ApplicationController
      def add_inventory_quantity
        @products.each do |product|
          /*Chama o serviço de inventário via API*/
          response = HTTParty.post('http://inventory-service/
                                   body: { product_id: produ
```

```
                                                quantity: product
                                                store_entrance: p
                                                account_id: produ
          /*Lida com a resposta da API conforme necessário*/
        end

        render json:
            {
            status: 'success', message: 'Inventory update ini
            }, status: :ok
      end
    end
  end
end
```