Análise da pasta: open-erp/app/models/services/bling

Arquivo: find_order.rb

Este código é uma implementação de um serviço em Ruby, seguindo o padrão de arquitetura “Service Object”.

```
# frozen_string_literal: true

module Services
  module Bling
    class FindOrder < ApplicationService
      attr_accessor :id, :order_command, :tenant

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

      def call
        case order_command
        when 'find_order'
          find_order
        else
          raise 'Not a order command'
        end
      end

      private
```

```
        def find_order
          token = bling_token
          url = URI("https://www.bling.com.br/Api/v3/pedidos/ve

          headers = {
            'Accept' => 'application/json',
            'Authorization' => "Bearer #{token}"
          }

          http = Net::HTTP.new(url.host, url.port)
          http.use_ssl = true

          request = Net::HTTP::Get.new(url, headers)

          response = http.request(request)

          data = JSON.parse(response.read_body)

          data
        rescue StandardError => e
          "Error: #{e.message}"
        end

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

Estrutura do código:

O código começa definindo dois módulos: `'Services'` e `'Bling'`, módulos estes que são utilizados para organizar o código e fornecer um namespace adequado. O uso de módulos ajuda a evitar colisões de nomes e a agrupar funcionalidades relacionadas.

Dentro do módulo `'Bling'`, temos a classe `'FindOrder'`, que herda de `'ApplicationService'`. A herança sugere que o serviço `FindOrder` é uma

especialização de uma classe base `ApplicationService`, que provavelmente contém lógica ou comportamentos comuns a todos os serviços da aplicação.

Além disso, a classe `"FindOrder"` possui três atributos acessores: `'id'`, `'order_command'` e `'tenant'`. Esses atributos são inicializados através do método `'initialize'`, que recebe um hash de argumentos nomeados.

O método `'call'` é o ponto de entrada do serviço. Verificando o valor de `'order_command'` e, se for igual a `'find_order'`, invoca o método `'find_order'`. Caso contrário, levanta uma exceção. Isso sugere que o serviço está preparado para lidar com diferentes comandos relacionados a pedidos (orders).

O método privado `'find_order'`, é responsável por realizar a busca de um pedido na API da **Bling**, ou seja, ele:

- Obtém o **token** de autenticação através do método `bling_token`.
- Monta a URL da API, substituindo `@id` pelo identificador do pedido.
- Configura os cabeçalhos da requisição, incluindo o token de autenticação.
- Faz uma requisição HTTP GET à API utilizando `Net::HTTP`.
- Lê e parseia a resposta da API como JSON e retorna os dados.

O método captura erros usando `'rescue'` e retorna uma string com a mensagem de erro em caso de falha.

Agora o método privado `'bling_token'`, este retorna o token de acesso necessário para autenticar a requisição à API da Bling. Ele utiliza o cache de instância `( '@bling_token ||=' )` para evitar consultas repetidas ao banco de dados.

Análise Arquitetural

Service Object Pattern:

A arquitetura utiliza o padrão **Service Object**, onde a lógica de negócios específica é encapsulada em classes de serviço. Isso ajuda a manter os modelos (e outras partes da aplicação) menos carregados de lógica complexa, facilitando a manutenção e a testabilidade.

Herança:

A herança de `‘ApplicationService’` sugere uma arquitetura onde a lógica comum entre diferentes serviços é centralizada em uma classe base, permitindo reutilização e DRY (Don't Repeat Yourself).

Modularidade:

A organização em módulos `(Services::Bling)` e a separação de responsabilidades (e.g., obtenção de tokens, requisições HTTP) demonstram uma preocupação com a modularidade e a separação de responsabilidades, facilitando a escalabilidade do código.

Manutenção e Extensibilidade:

A utilização de `‘case’` no método `‘call’` indica que a classe pode ser facilmente estendida para lidar com novos comandos relacionados a pedidos, mantendo o código flexível e extensível.

Segurança e Tratamento de Erros:

O tratamento de erros com `‘rescue’` dentro do método `‘find_order’` é uma boa prática para evitar que exceções quebrem o fluxo da aplicação. No entanto, o retorno de uma string com a mensagem de erro pode ser uma abordagem simplista. Dependendo do contexto, seria interessante levantar uma exceção personalizada ou retornar um objeto de erro estruturado.

Uso de HTTP e Dependências Externas:

A classe faz uso direto de `‘Net::HTTP’` para realizar requisições HTTP. Embora isso funcione, em uma aplicação maior, talvez seja interessante encapsular essa lógica em uma camada dedicada para facilitar a manutenção e substituição de clientes HTTP, caso necessário.

Arquivo: find_orders.rb

O código é uma extensão do serviço anterior, que lida com múltiplos pedidos em vez de apenas um.

```ruby
# frozen_string_literal: true

module Services
  module Bling
    class FindOrders < ApplicationService
      attr_accessor :ids, :order_command, :tenant

      def initialize(order_command:, tenant:, ids:)
        @order_command = order_command
        @tenant = tenant
        @ids = ids
      end

      def call
        case order_command
        when 'find_orders'
          find_orders
        else
          raise 'Not a order command'
        end
      end

      private

      def find_orders
        # Initialize hash for counting and a counter variable
        codigo_quantidade = {}
        counter = 0

        # Loop through each ID
        @ids.each do |id|
          # Call the service for each ID
          response = Services::Bling::FindOrder.call(id: id, 
          next unless response && response["data"] && respons

          # Process each item
          response["data"]["itens"].each do |item|
            order = BlingOrderItem.find_by(bling_order_id: id
```

```
            order.update(items: item) if order.present?
            codigo = item["codigo"]
            quantidade = item["quantidade"]

            # Update counts
            codigo_quantidade[codigo] = codigo_quantidade.fet

            # Increment the counter
            counter += 1

            # Print the current item's codigo and quantidade
            puts "Item #{counter} - Codigo: #{codigo}, Quanti
          end
        end

        # File path for the CSV
        csv_file_path = 'codigo_quantidade.csv'

        # Generate CSV
        CSV.open(csv_file_path, 'w') do |csv|
          # Adding headers to the CSV file
          csv << ['Codigo', 'Total Quantidade']

          # Writing data to the CSV file
          codigo_quantidade.each do |codigo, quantidade|
            csv << [codigo, quantidade]
          end
        end

        puts "CSV file generated: #{csv_file_path}"

      end
    end
  end
end
```

Estrutura do código:

Como no código anterior, o serviço está organizado dentro dos módulos `'Services'` e `'Bling'`, que mantém uma estrutura modular e um namespace claro.

Na classe `'FindOrders'` herda de `'ApplicationService'`, seguindo a mesma lógica de especialização de serviços que a classe `'FindOrder'`. Agora a classe lida com múltiplos pedidos, conforme evidenciado pelo atributo `'ids'`, que é um array de IDs.

Os atributos acessores incluem `'ids'`, `'order_command'`, e `'tenant'`. Eles são inicializados através do método `'initialize'`, que recebe um hash de argumentos nomeados.

Já o método `'call'` atua como o ponto de entrada para o serviço, verificando o valor de `'order_command'`. Se for `'find_orders'`, o método `'find_orders'` é chamado. Em caso contrário, uma exceção é levantada.

No método privado `'finde_orders'` itera sobre cada ID em `'@ids'`, chamado o serviço `'find_orders'` para buscar os dados de cada pedido.

Em seguida, processa os itens de cada pedido, atualizando ou criando registros na base de dados, e mantém um contador e um hash `( 'codigo_quantidade' )` para somar as quantidades de itens com base no seu código.

Quanto à atualização de registro, o código verifica se já existe um item de pedido `( 'BlingOrderItem' )` que corresponde ao `'bling_order_id'` e, se existir, atualiza os itens com os dados obtidos da API Bling.

Também, para cada item processado, o código extrai o `'codigo'` e a `'quantidade'`, incrementando a quantidade correspondente no hash `'codigo_quantidade'`.

Após processar todos os IDs, o método cria um arquivo CSV (codigo_quantidade.csv) que contém o total das quantidades agrupadas por código. Ao longo do processo, há saídas para o console com detalhes de cada item processado e uma mensagem final confirmando a geração do arquivo CSV.

Análise Arquitetural:

Service Object Pattern:

Assim como o código anterior, este serviço segue o padrão **Service Object**, encapsulando uma operação específica (neste caso, buscar e processar múltiplos pedidos) em uma classe dedicada.

Modularidade e Reuso:

O serviço `‘FindOrders’` reutiliza o serviço `‘FindOrder'` para buscar os dados de cada pedido individual. Isso promove o reuso de código e mantém as responsabilidades claramente definidas entre os serviços.

A modularidade é reforçada com o uso de módulos e métodos privados, facilitando a manutenção e escalabilidade do código.

Manutenção e Extensibilidade:

O código é extensível no sentido de que novos comandos ou funcionalidades poderiam ser adicionados com relativa facilidade ao serviço, assim como foi feito para o comando `'find_orders'`.

A modularidade permite que mudanças ou melhorias em um serviço (como FindOrder) sejam refletidas no outro serviço (FindOrders), sem a necessidade de duplicar lógica.

Tratamento de Erros:

O método `‘find_orders’` não possui um tratamento explícito de erros. Se uma requisição falhar ou se houver algum problema ao processar os dados, o código pode levantar exceções inesperadas ou produzir saídas inconsistentes. Adicionar blocos de rescue ou validações mais robustas pode melhorar a resiliência do código.

Desempenho:

O serviço está iterando sobre uma lista potencialmente grande de **IDs** e processando cada um individualmente, o que pode ser custoso em termos de desempenho. Dependendo do tamanho da lista e da quantidade de itens em cada pedido, o tempo de execução pode ser significativo.

A lógica de contar os itens **(codigo_quantidade)** em tempo real e a geração de um CSV ao final indicam uma operação síncrona. Para grandes volumes de

dados, pode ser interessante considerar uma abordagem assíncrona ou a utilização de background jobs.

Uso de Dependências e Saída de Dados:

O uso direto de **puts** para saída de dados pode não ser ideal em um ambiente de produção. Em vez disso, a saída poderia ser registrada em logs, que podem ser monitorados e analisados posteriormente.

A geração do CSV é uma boa prática para persistir os resultados, mas o caminho do arquivo ('codigo_quantidade.csv') está fixo e poderia ser parametrizado para maior flexibilidade.

Integração com Bancos de Dados:

O código faz uso da classe `'BlingOrderItem'` para verificar e atualizar registros no banco de dados. Isso sugere que a aplicação está integrada com um ORM (provavelmente ActiveRecord, dado o contexto Rails), o que é uma prática comum e eficiente para gerenciar interações com o banco de dados.

Assim como o outro código, este também segue o padrão **Service Object**, encapsulando uma operação específica, que neste caso, seria buscar e processar múltiplos pedidos, em uma classe dedicada.

Arquivo: order.rb

```
# frozen_string_literal: true

module Services
  module Bling
    class Order < ApplicationService
      attr_accessor :order_command, :tenant, :situation, :opt

      def initialize(order_command:, tenant:, situation:, opt
        @order_command = order_command
        @tenant = tenant
        @situation = situation
        @options = options
        @max_pages = options[:max_pages]
```

```
        options.delete(:max_pages)
      end

      def call
        case order_command
        when 'find_orders'
          find_orders
        else
          raise 'Not an order command'
        end
      end

      private

      def find_orders
        token = bling_token
        base_url = 'https://www.bling.com.br/Api/v3/pedidos/v
        params = {
          limite: 100,
          idsSituacoes: [situation]
        }

        params.merge!(options)

        headers = {
          'Accept' => 'application/json',
          'Authorization' => "Bearer #{token}"
        }

        all_orders = []

        (1..@max_pages).each do |page|
          # we do not need to request all massive data. If th
          break if (page.eql?(2) && Rails.env.eql?('test'))

          response = HTTParty.get(base_url, query: params.mer
          if response['error'].present?
            sleep 10
```

```
            response = HTTParty.get(base_url, query: params.m
          end

          raise(StandardError, response['error']['type']) if

          data = JSON.parse(response.body)
          break if data['data'].blank?

          all_orders.concat(data['data'])
        end

        { 'data' => all_orders }
      end

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

Estrutura do código:

Semelhante aos anteriores, utilizando o padrão **Service Object**.

Também organizado dentro dos módulos `'Services'` e `'Bling'`, criando uma estrutura de namespace clara e modular, o que mantém o código organizado facilitando a localização de funcionalidades específicas.

A classe `Order` herda de `ApplicationService`, seguindo a lógica de especialização de serviços que vimos nas implementações anteriores. A classe encapsula a lógica de operações relacionadas a pedidos (orders) na API Bling.

Os atributos acessores incluem `order_command`, `tenant`, `situation`, e `options`. Eles são inicializados através do método initialize, que recebe um hash de argumentos nomeados.

O atributo `@max_pages` é extraído das opções passadas e armazenado separadamente. A chave `:max_pages` é removida de options para que não interfira nos parâmetros de requisição.

No método `‘call’` verifica o valor de `‘order_command’` e caso for o `‘find_orders’`, ele invoca o método `‘find_orders’`, se não for, uma exceção é levantada. Este padrão de roteamento de comandos permite a extensibilidade do serviço para suportar novas funcionalidades.

O método privado `‘find_orders’` é responsável por buscar pedidos da API **Bling**, ou seja:

- Obtém o **token** de autenticação através do método `bling_token`**.**
- Define a URL base e os parâmetros da requisição, incluindo a situação dos pedidos e quaisquer opções adicionais.
- Configura os cabeçalhos da requisição, especialmente o **token** de autenticação.
- Faz requisições paginadas à API usando **HTTParty** para obter todas as páginas de resultados até o limite definido em `@max_pages`**.**
- Em caso de erro, há uma tentativa de reenvio após um **‘sleep’** de 10 segundos.
- Se o dado retornado estiver vazio, a iteração para, presumivelmente porque não há mais páginas a serem retornadas.
- O método acumula todos os pedidos retornados em `‘all_orders’` e os retorna como um hash com a chave `'data'`.

Já o método privado `‘Bling Token’` obtém o token de autenticação para a API Bling, que utiliza uma consulta ao banco de dados para encontrar o `‘access_token’` associado ao `‘tenant’` (identificado pela **‘account_id’**).

Arquitetura do código

Service Object Pattern:

A arquitetura segue o padrão **Service Object**, onde a lógica de negócios específica é encapsulada em uma classe de serviço. Isso mantém a aplicação modular e facilita a manutenção e testabilidade.

Modularidade e Reuso:

A estrutura modular facilita o reuso e a extensão do serviço. O método **call** é configurado para lidar com diferentes comandos `(order_command)`, permitindo a

adição de novas funcionalidades no futuro sem modificar a estrutura existente.

A separação da lógica de obtenção do **token** em um método separado `(bling_token)` também exemplifica o reuso e a modularidade.

Manutenção e Extensibilidade:

O código é projetado para ser facilmente extensível, especialmente através do padrão de comandos no método `call`. Novos comandos podem ser adicionados conforme necessário.

A iteração paginada é uma solução escalável para lidar com grandes volumes de dados, embora o uso de HTTParty diretamente poderia ser encapsulado para facilitar a substituição ou ajustes no cliente HTTP.

Tratamento de Erros:

O método find_orders possui algum nível de tratamento de erros, incluindo uma lógica de retry (sleep 10 e nova requisição em caso de erro). No entanto, a maneira como os erros são levantados `(raise(StandardError, response['error']['type']))` pode ser aprimorada, talvez utilizando exceções personalizadas para melhor contexto e manuseio de erros no código chamador.

Desempenho e Escalabilidade:

A implementação considera a limitação de páginas `(@max_pages)`, o que é importante para controlar o volume de dados e o tempo de execução. Também há uma quebra antecipada no loop de iteração de páginas em um ambiente de teste, o que sugere uma consideração cuidadosa para cenários de teste.

A decisão de parar a iteração ao encontrar uma página vazia é uma otimização que previne requisições desnecessárias, melhorando o desempenho.

Uso de Dependências e Saída de Dados:

O código depende de HTTParty para fazer as requisições HTTP. Enquanto isso é funcional, encapsular essa lógica em um serviço dedicado pode facilitar a substituição futura do cliente HTTP ou a adição de funcionalidades como logging detalhado ou cacheamento.

A ausência de puts ou outras formas de output sugere que o serviço está projetado para ser utilizado dentro de outros componentes da aplicação, sem necessidade de saída direta para o console, o que é uma prática adequada em serviços de backend.

A arquitetura deste código é sólida e segue o padrão **Service Object**, o que promove modularidade, reuso e extensibilidade. O serviço é bem organizado, com preocupações claras em relação ao desempenho (paginação, retry) e tratamento básico de erros.

Arquivo: product.rb

```
# frozen_string_literal: true

module Services
  module Bling
    # Service::Bling::Product is used in the GoodJob service
    # Model Product.synchronize_bling uses it to create/updat
    class Product < ApplicationService
      attr_accessor :product_command, :tenant, :options

      def initialize(product_command:, tenant:, options: {})
        @product_command = product_command
        @tenant = tenant
        @options = options
        @max_pages = options[:max_pages]
        options.delete(:max_pages)
      end

      def call
        case product_command
        when 'find_products'
          find_products
        else
          raise 'Not a product command'
        end
      end
```

```
    private

    def find_products
      token = bling_token
      base_url = 'https://www.bling.com.br/Api/v3/produtos'
      params = {
        limite: 100
      }

      params.merge!(options)

      headers = {
        'Accept' => 'application/json',
        'Authorization' => "Bearer #{token}"
      }

      all_products = []

      (1..@max_pages).each do |page|
        # we do not need to request all massive data. If th
        break if (page.eql?(2) && Rails.env.eql?('test'))

        response = HTTParty.get(base_url, query: params.mer
        sleep 5 if Rails.env.eql?('production') || Rails.en
        if response['error'].present?
          sleep 10 if Rails.env.eql?('production') || Rails
          response = HTTParty.get(base_url, query: params.m
        end

        if response['error'].present?
          sleep 10 if Rails.env.eql?('production') || Rails
          response = HTTParty.get(base_url, query: params.m
        end

        raise(StandardError, response['error']['type']) if

        data = JSON.parse(response.body)
```

```
          break if data['data'].blank?

          all_products.concat(data['data'])
        end

        { 'data' => all_products }
      end

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

Organizado nos módulos `'Service'` e `'Bling'`, o código estabelece uma clara separação de responsabilidades, mantendo-o modular e fácil de navegar. Essa estrutura facilita a manutenção e a escalabilidade da aplicação.

A classe `'Product'`, ao herdar de `'ApplicationService'`, indica que faz parte de um conjunto de serviços especializados com estrutura e interface consistentes. Projetada para interagir com a API de produtos do Bling, ela permite buscar produtos e realizar operações específicas baseadas em comandos.

Já os atributos acessores incluem `‘product_command’`, `‘tenant’`, e `‘options’`. Eles são inicializados através do método `initialize`, que recebe um hash de argumentos nomeados.
O atributo
`‘@max_pages’` é extraído das `‘options’` e removido do hash original para evitar conflitos nos parâmetros da requisição.

O método `‘call’` serve como ponto de entrada para o serviço, executando a lógica com base no valor de `‘ product_command’`. Se o comando for `‘find_products’`, o método `‘ find_products’` é invocado; caso contrário, uma exceção é lançada. Este padrão de comando proporciona uma arquitetura extensível, permitindo a adição de novos comandos conforme necessário.

Como o método privado `‘find_products’` é responsável por buscar os produtos na API Bling, ele tem a seguinte lógica:

- Obtém o token de autenticação via `bling_token`.
- Define a URL base e os parâmetros para a requisição, incluindo limites e outras opções passadas.
- Configura os cabeçalhos da requisição, especialmente o token de autenticação.
- Faz requisições à API de forma paginada, iterando através de um loop que vai até `@max_pages`. A resposta é processada e, caso ocorra um erro, o método tenta novamente após um intervalo (`sleep`).
- Se um erro é detectado na resposta, o código tenta novamente, e se o erro persistir, ele é levantado como uma exceção.
- Os dados de cada página são acumulados em `all_products`, que é retornado ao final.

O método privado 'bling_token' é responsável por obter o token de autenticação para a API Bling. Ele utiliza uma consulta ao banco de dados (`BlingDatum.find_by(account_id: @tenant)`) e armazena o resultado em uma variável de instância (`@bling_token`). Essa abordagem evita consultas repetidas ao banco de dados, melhorando a eficiência do código.

Análise Arquitetural

Service Object Pattern:

A arquitetura segue o padrão **Service Object**, onde a lógica de negócio é encapsulada dentro de um serviço específico. Isso permite que o código seja modular, reutilizável e fácil de testar.

Modularidade e Reuso:

A modularidade é evidente na separação entre o serviço `(Product)` e a lógica de obtenção de tokens `(bling_token)`. Isso facilita a manutenção e o reuso do código.

A implementação do método **call** usando o padrão de comando `(product_command)` permite a fácil extensão do serviço para suportar novos tipos de operações sem necessidade de reescrever o código existente.

Manutenção e Extensibilidade:

A classe está bem estruturada para suportar a adição de novos comandos no futuro. Se houver necessidade de expandir as funcionalidades relacionadas a produtos do Bling, novos comandos podem ser adicionados no método call sem modificar a lógica existente.

A reutilização de HTTParty e o padrão de paginação permitem que o código funcione de maneira eficiente, mesmo com grandes volumes de dados.

Tratamento de Erros:

O código inclui mecanismos básicos de retry em caso de erros, com intervalos `(sleep)` entre as tentativas. Embora isso forneça alguma resiliência, o tratamento de erros poderia ser melhorado com a introdução de exceções customizadas ou com o uso de mecanismos de logging para registrar detalhes sobre falhas.

Desempenho e Escalabilidade:

O código considera a limitação de páginas `(@max_pages)`, o que ajuda a controlar o volume de dados e o tempo de execução. A introdução de intervalos `(sleep)` em ambientes de produção e staging indica uma preocupação com o desempenho e a escalabilidade, evitando sobrecarregar a API do Bling.

A lógica para quebrar o loop de iteração em ambientes de teste `(Rails.env.eql?('test'))` sugere uma abordagem cuidadosa para testes, permitindo uma execução mais rápida em cenários controlados.

Dependências e Saída de Dados:

O código depende de HTTParty para fazer as requisições HTTP. Embora essa biblioteca seja amplamente utilizada, encapsular a lógica de requisição dentro de um método ou serviço dedicado pode oferecer maior flexibilidade e a possibilidade de adicionar funcionalidades como cacheamento ou retries mais robustos.

A ausência de saídas diretas (puts) sugere que o serviço é destinado a ser utilizado como parte de uma pipeline maior, o que é uma prática adequada em serviços backend.

Arquivo: stock.rb

Segue a mesma linha dos outros códigos acima, utilizando a arquitetura **Service Object**, que organiza a lógica de negócio associada à integração com a API do Bling.

```
# frozen_string_literal: true

module Services
  module Bling
    # Service::Bling::Stock is used in the GoodJob service to
    # Model Stock.synchronize_bling uses it to create/update
    class Stock < ApplicationService
      attr_accessor :stock_command, :tenant, :options

      def initialize(stock_command:, tenant:, options: {})
        @stock_command = stock_command
        @tenant = tenant
        @options = options
        options.delete(:max_pages)
      end

      def call
        case stock_command
        when 'find_stocks'
          find_stocks
        else
          raise 'Not a stock command'
        end
      end

      private

      def find_stocks
        token = bling_token
        base_url = 'https://www.bling.com.br/Api/v3/estoques/
        params = {}

        params.merge!(options)

        headers = {
          'Accept' => 'application/json',
          'Authorization' => "Bearer #{token}"
        }
```

```
        all_stocks = []
        response = HTTParty.get(base_url, query: params, head

        raise(StandardError, response['error']['type']) if re

        data = JSON.parse(response.body)

        all_stocks.concat(data['data'])

        { 'data' => all_stocks }
      end

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

Estrutura do Código

O código está estruturado nos módulos `Services` e `Bling`, organizando eficientemente a lógica de negócio e separando as responsabilidades. Essa organização aprimora a navegabilidade e facilita a manutenção do código, além de permitir a reutilização de componentes em várias partes da aplicação.

A classe `Stock`, ao herdar de `ApplicationService`, indica sua participação em uma arquitetura de serviços onde classes especializadas herdam comportamentos comuns e utilitários de uma classe base.

Projetada para gerenciar operações de estoque, a classe utiliza um comando (`stock_command`) para determinar a ação a ser executada, seguindo um padrão similar ao de outras classes de serviço.

Os atributos da classe incluem `stock_command`, `tenant` e `options`. O método `initialize` recebe esses valores como argumentos nomeados e os inicializa.

No `initialize`, há um detalhe importante: a opção `max_pages` é removida de `options`. Isso indica que este serviço específico não trabalha com paginação, diferentemente dos exemplos anteriores.

O método `call` serve como ponto de entrada do serviço. Ele verifica o valor de `stock_command` e executa a lógica correspondente. Para o comando `'find_stocks'`, o método privado `find_stocks` é invocado; caso contrário, uma exceção é lançada.

Essa estrutura é flexível e extensível, permitindo a adição de novos comandos conforme necessário.

O método `find_stocks` é responsável por buscar informações de estoque na API Bling. Sua sequência lógica é:

- Obtém o token via `bling_token`.
- Define a URL base (`base_url`) e os parâmetros (`params`) para a requisição, combinando-os com as opções fornecidas ao serviço.
- Configura os cabeçalhos da requisição, incluindo o token de autenticação.
- Utiliza `HTTParty` para fazer uma requisição GET à API Bling.
- Se a resposta contém um erro (`response['error']`), lança uma exceção com o tipo de erro específico.
- Extrai os dados retornados e os armazena na variável `all_stocks`, que é retornada ao final do método.

O método `bling_token` obtém o token de autenticação para acessar a API Bling. Ele consulta o banco de dados pelo `account_id` correspondente ao `tenant` atual e armazena o token em uma variável de instância (`@bling_token`), evitando múltiplas consultas ao banco.

Análise Arquitetural

Service Object Pattern:

O código adere ao padrão **Service Object**, que encapsula uma unidade de lógica de negócio em uma classe de serviço. Isso promove modularidade, reutilização, e uma interface clara para a execução de operações complexas.

Modularidade e Reuso:

A classe `Stock` é modular, isolando a lógica relacionada ao estoque da API Bling em um serviço específico. Isso facilita a manutenção e permite que outros serviços ou partes da aplicação utilizem essa lógica sem duplicação de código.

Extensibilidade:

O design da classe é extensível. Novos comandos de estoque podem ser facilmente adicionados ao método `call`, permitindo que a classe suporte operações adicionais sem necessidade de refatoração significativa.

Tratamento de Erros:

O código inclui um tratamento básico de erros, levantando exceções se a resposta da API Bling contiver um erro. No entanto, não há tentativas de reexecutar a requisição em caso de falhas temporárias, o que poderia ser uma melhoria.

Performance e Escalabilidade:

Diferente dos serviços anteriores que manipulavam grandes volumes de dados paginados, este serviço não inclui lógica de paginação, o que pode indicar que o volume de dados esperado é menor ou que a API específica de estoque não suporta paginação.

Dependências e Manutenção:

O uso de `HTTParty` para fazer requisições HTTP é uma escolha comum, mas, como mencionado anteriormente, encapsular essa lógica em um método ou serviço dedicado poderia aumentar a flexibilidade, permitindo melhorias como cacheamento ou um sistema de retries mais sofisticado.

A separação clara entre o serviço e o mecanismo de autenticação (o método `bling_token`) é uma boa prática, pois permite a modificação ou substituição fácil da lógica de autenticação sem impactar outras partes do código.

Arquivo: update_order_status.rb

Segue a mesma arquitetura dos códigos anteriores, utiliza **Service Object Pattern** que serve para encapsular a lógica de negócio relacionada à atualização de status de pedidos na API do Bling.

```
# frozen_string_literal: true
```

```
module Services
  module Bling
    class UpdateOrderStatus < ApplicationService
      BASE_URL = 'https://www.bling.com.br/Api/v3/pedidos/ven

      attr_accessor :tenant, :order_ids, :new_status

      def initialize(tenant:, order_ids:, new_status:)
        @tenant = tenant
        @order_ids = order_ids
        @new_status = new_status
      end

      def call
        token = bling_token
        base_url = 'https://www.bling.com.br/Api/v3'
        results = []

        order_ids.each do |order_id|
          url = "#{BASE_URL}#{order_id}/situacoes/#{new_statu
          response = HTTParty.patch(url, headers: authorizati

          results << if response.success?
                       { order_id: order_id, status: 'success
                     else
                       { order_id: order_id, status: 'failed'
                     end
        end

        { results: results }
      end

      private

      def bling_token
        BlingDatum.find_by(account_id: tenant).access_token
      end
```

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

Também organizado nos módulos **'Services'** e **'Bling'**, como os anteriores, o que organiza o código e facilita a localização e manutenção de funcionalidades específicas.

A classe **'UpdateOrderStatus'** herda de **'ApplicationService'**, indicando sua participação em um conjunto de serviços que compartilham funcionalidades comuns definidas na classe base. Esta classe tem como responsabilidade específica a atualização do status de um ou mais pedidos na API do Bling.
A URL base para as requisições à API de pedidos do Bling é armazenada em uma constante (
`BASE_URL`). Isso facilita a manutenção e evita a repetição de strings de URL ao longo do código.

O método
'call' é o ponto de entrada do serviço e executa as seguintes ações:

- Obtém o token de autenticação via método `bling_token`.
- Inicializa uma lista `results` para armazenar os resultados de cada operação.
- Itera sobre os `order_ids`, construindo a URL específica e enviando uma requisição PATCH à API para atualizar o status de cada pedido.
- Registra o resultado de cada atualização em `results`, indicando sucesso ou falha, incluindo mensagens de erro quando aplicável.
- Retorna um hash contendo todos os resultados processados.

Nós também temos mais dois métodos privados: 'bling_token' e 'authorization_headers'. Um obtém o token de autenticação necessário para acessar a API Bling, consultando o banco de dados com base no `tenant,`

e o outro retorna os cabeçalhos de autorização que serão incluídos em cada requisição, utilizando o token obtido.

Análise Arquitetural

Service Object Pattern:

A arquitetura do código segue o **Service Object Pattern**, encapsulando a lógica de atualização de status de pedidos em uma classe de serviço específica. Esse padrão promove a separação de responsabilidades e a modularidade do código.

Modularidade e Reuso:

A classe `UpdateOrderStatus` é bem encapsulada, isolando a lógica de negócio relacionada à atualização de status de pedidos na API do Bling. Esse isolamento facilita a reutilização do código em diferentes partes da aplicação e torna a manutenção mais simples.

Extensibilidade:

A classe foi projetada de forma extensível. Por exemplo, se fosse necessário suportar novos tipos de comandos ou diferentes operações de atualização, seria possível expandir a classe sem a necessidade de uma refatoração significativa.

Tratamento de Erros:

O código inclui um tratamento básico de erros. Se uma requisição falhar, a resposta falhada é registrada no array `results`, junto com uma mensagem de erro. No entanto, não há uma lógica de retries ou reexecução em caso de falhas temporárias, o que poderia ser uma melhoria.

Consistência e Manutenção:

A utilização de uma constante (`BASE_URL`) para a URL base da API melhora a consistência e a manutenção do código. No entanto, a URL `base_url` inicializada dentro do método `call` parece não ser utilizada, o que pode indicar um pequeno ponto a ser revisado.

A separação do método `authorization_headers` ajuda a manter o código limpo e fácil de entender, evitando a repetição de lógica comum a todas as requisições.

Dependências:

O código depende da gem `HTTParty` para realizar as requisições HTTP, uma escolha comum e adequada para essa tarefa. O uso de `HTTParty` é direto e eficiente para esse caso de uso específico.