

Universidade de Coimbra

Departamento de Engenharia Informática



Sistemas de Gestão de Dados

Relatório Final do Projeto Pet Store



Trabalho Realizado por:

Catarina Cruz - 2020240694

Inês Barata - 2021200507

Mafalda Duarte - 2021236492

2023/2024

Introdução

No âmbito da unidade curricular de Sistemas de Gestão de Dados foi-nos proposta a realização de um projeto cujo objetivo é criar um sistema de gestão de dados de uma loja de animais. Para isso precisamos de criar uma base de dados, um modelo ER, as tabelas da base de dados e processar pedidos HTTP através da REST API.

Através deste projeto pretendemos aprofundar os nossos conhecimentos sobre a organização e implementação de uma aplicação de base de dados.

Desenvolvimento do Modelo ER

Na primeira fase do projeto, criamos um modelo Entidade-Relacionamento (ER) e o respetivo Modelo Físico. Posteriormente, com a avaliação do docente, percebemos que este apresentava alguns problemas, como por exemplo algumas entidades fracas que precisavam de ser definidas, e decidimos então realizar algumas mudanças.

Assim, apresentamos a versão corrigida nas figuras 1 e 2 .

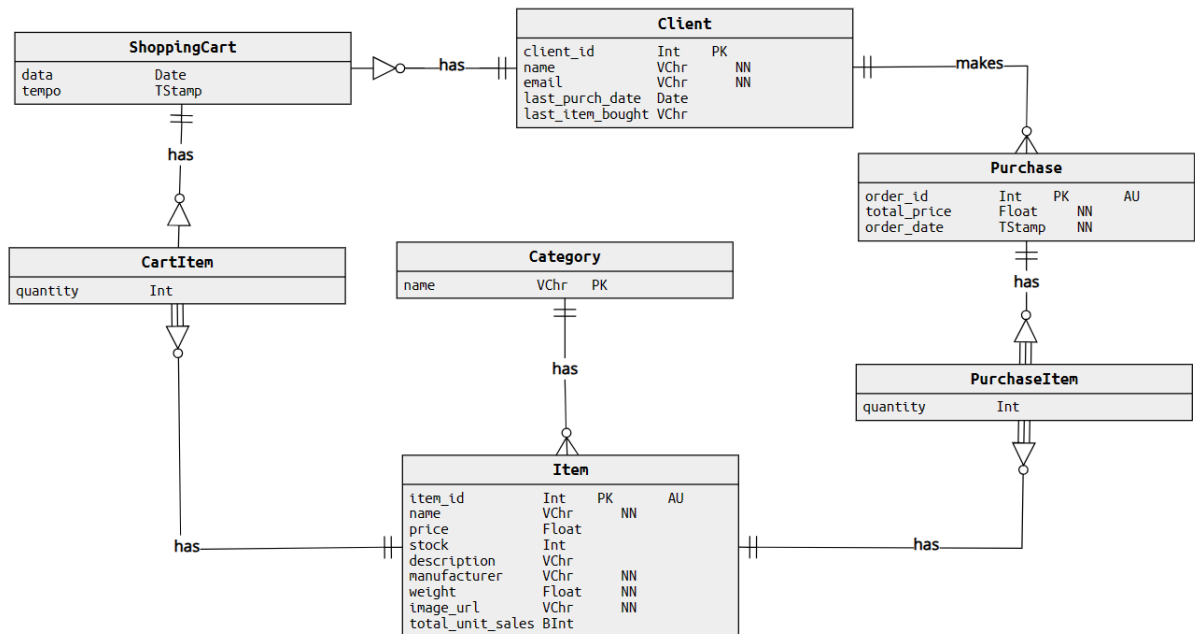


Fig.1 - Modelo ER Final

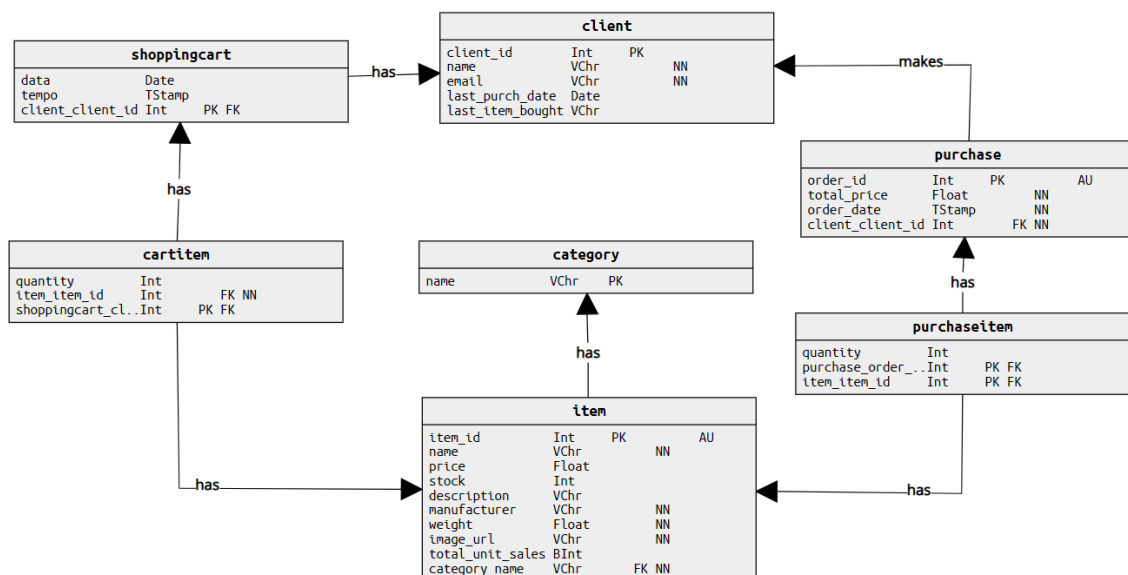


Fig.2 - Modelo Físico Final

Estrutura Geral do Projeto

Projeto:

→ *load_data.py*

- ◆ *query ()*
- ◆ *db_connection()*
- ◆ código de criação de tabelas e load dos dados

→ *api.py*

- ◆ *landing_page()*
- ◆ *create_item()*
- ◆ *update_item()*
- ◆ *delete_item_from_cart()*
- ◆ *add_item_to_cart()*
- ◆ *get_items_list()*
- ◆ *get_item_details()*
- ◆ *search_items()*
- ◆ *get_top_sales_per_category()*
- ◆ *purchase_items()*
- ◆ *get_clients_with_filters()*
- ◆ *add_client()*
- ◆ *get_client_orders()*

Carregamento dos Dados

O carregamento dos dados foi feito no ficheiro *load_data.py*, mas antes tivemos de criar algumas funções e tabelas como vamos passar a explicar.

A primeira função que definimos foi a *query()* que vai ser utilizada para executar comandos SQL. A seguir criámos a *db_connection()* que vai estabelecer uma ligação a uma base de dados no PostgreSQL de acordo com os dados definidos. Estas duas funções irão auxiliar-nos na realização das próximas etapas.

O passo seguinte consistiu na criação das tabelas com base no nosso diagrama. Criámos um total de 7 tabelas, todas elas com um código semelhante ao apresentado na figura 3.

```
CREATE TABLE item (  
    item_id SERIAL PRIMARY KEY,  
    name VARCHAR(512) NOT NULL,  
    category VARCHAR(512) REFERENCES category(name),  
    price REAL,  
    stock INTEGER,  
    description VARCHAR(512),  
    manufacturer VARCHAR(512) NOT NULL,  
    weight REAL NOT NULL,  
    image_url VARCHAR(512) NOT NULL,  
    total_unit_sales INTEGER  
)
```

Fig.3 - Código para criar a tabela *item*

No final da criação das tabelas procedemos à adição das restrições de foreign keys, como podemos observar na figura 4, de maneira a garantirmos a consistência dos dados.

```
ALTER TABLE item ADD CONSTRAINT item_fk1 FOREIGN KEY (category) REFERENCES category(name);  
ALTER TABLE purchase ADD CONSTRAINT purchase_fk1 FOREIGN KEY (client_client_id) REFERENCES client(client_id);  
ALTER TABLE shoppingcart ADD CONSTRAINT shoppingcart_fk1 FOREIGN KEY (client_client_id) REFERENCES client(client_id);  
ALTER TABLE cartitem ADD CONSTRAINT cartitem_fk1 FOREIGN KEY (item_item_id) REFERENCES item(item_id);  
ALTER TABLE cartitem ADD CONSTRAINT cartitem_fk2 FOREIGN KEY (shoppingcart_client_client_id) REFERENCES shoppingcart(client_client_id);  
ALTER TABLE purchaseitem ADD CONSTRAINT purchaseitem_fk1 FOREIGN KEY (purchase_order_id) REFERENCES purchase(order_id);  
ALTER TABLE purchaseitem ADD CONSTRAINT purchaseitem_fk2 FOREIGN KEY (item_item_id) REFERENCES item(item_id);
```

Fig.4- Restrições das Foreign Keys

Já com as tabelas criadas podemos então carregar os dados, para isto criamos dados para cada uma de acordo com os parâmetros necessários e com o cuidado de respeitar as Fk (foreign keys). A figura 5 mostra um exemplo referente aos dados para a tabela *item*.

```

# ITEM table -----
items_data = [
    (1246, 'Premium Dog Bed', 'Accessories', 49.99, 100, 'Luxury bed for dogs', 'ComfyPets', 3.0, 'https://example.com/item-dog-bed.jpg', 5),
    (1537, 'Cat Tunnel', 'Toys', 19.99, 200, 'Interactive tunnel for cats', 'PlayfulPets Inc.', 1.2, 'https://example.com/item-cat-tunnel.jpg', 8),
    (1348, 'Pet Grooming Gloves', 'Accessories', 12.49, 150, 'Gloves for grooming pets', 'GroomingPro', 0.5, 'https://example.com/item-grooming-gloves.jpg', 12),
    (1449, 'Organic Dog Treats', 'Food', 8.99, 300, 'Healthy treats for dogs', 'OrganicTreats', 0.8, 'https://example.com/item-dog-treats.jpg', 15),
    (2240, 'Feather Teaser Toy', 'Toys', 5.49, 180, 'Feather teaser toy for cats', 'FeatherPlay', 0.3, 'https://example.com/item-feather-toy.jpg', 10),
    (1321, 'Large Dog Crate', 'Accessories', 79.99, 150, 'Spacious crate for large dogs', 'PetHaven', 10.0, 'https://example.com/item-dog-crate.jpg', 3),
    (2242, 'Interactive Laser Toy', 'Toys', 14.99, 120, 'Automatic laser toy for cats', 'LaserPlay', 0.5, 'https://example.com/item-laser-toy.jpg', 7),
    (1423, 'Adjustable Cat Harness', 'Accessories', 18.49, 180, 'Harness for walking cats', 'WalkSafe', 0.8, 'https://example.com/item-cat-harness.jpg', 5),
    (2324, 'Large Bag of Bird Seed', 'Food', 11.99, 250, 'Nutritious seed mix for birds', 'FeatherFeast', 1.0, 'https://example.com/item-bird-seed.jpg', 12),
    (1425, 'Durable Dog Chew Toy', 'Toys', 9.79, 200, 'Long-lasting chew toy for dogs', 'ChewMaster', 0.7, 'https://example.com/item-chew-toy.jpg', 8),
    (1821, 'Dei Acc', 'Accessories', 92.79, 150, 'Ouf Ouf Miao Miao', 'DEiPet', 0.7, 'https://example.com/item-dei-toy.jpg', 8),
]

for row in items_data:
    #print(f'Inserting item data: {row}')
    query(conn, '''
        INSERT INTO item (item_id, name, category, price, stock, description, manufacturer, weight, image_url, total_unit_sales)
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    ''', row)

```

Fig.5 - Criação dos dados e carregamento para a tabela *item*

Implementação da API REST

Para a implementação da API REST, criamos 12 endpoints, cada um com as suas próprias funcionalidades e métodos HTTP correspondentes. A Tabela seguinte demonstra-o.

	Funcionalidade	Método HTTP	Endpoint
create_item()	Criar um novo item.	POST	/items
update_item(item_id)	Atualizar detalhes de um determinado item	PUT	/items/{item_id}
delete_item_from_cart(client_id, item_id)	Remover determinado item no carrinho específico.	DELETE	/carts/{client_id}/items/{item_id}
add_item_to_cart(client_id)	Adicionar determinado item.	POST	/cart/{client_id}
get_items_list()	Obter uma lista de todos os itens disponíveis.	GET	/items?page={n_page}&pageSize={limit}&sort={type_sort}&category={category}
get_item_details(item_id)	Obter detalhes de um determinado item.	GET	items/{item_id}
search_items(search)	Procurar determinado item.	GET	items/search/{name_item}
get_top_sales_per_category()	Obter as 3 maiores vendas de cada categoria.	GET	stats/sales
purchase_items()	Realizar uma compra.	POST	/purchase
get_clients_with_filters()	Obter uma lista com todos os clientes com base em critérios.	GET	/clients?last_purchase_date={data}&item_bought={item_name}
add_client()	Adicionar um cliente.	POST	/clients
get_client_orders(client_id)	Obter uma lista com todos os pedidos de determinado cliente.	GET	/clients/{client_id}/orders

Tabela 1 - Finalidades dos 12 endpoints.

Todos eles foram construídos sob a presença de *try-except* e operações como *commit* e *rollback*, reconhecendo a possibilidade de ocorrerem erros durante a execução destas operações. Para nos ajudar a reconhecer qual o erro está a acontecer, para além das mensagens personalizadas em cada função, retornamos o tipo de erro, sendo eles: 500 (erro interno relacionado com o servidor), 400 (o pedido não foi bem feito, por exemplo falta de parâmetros dados como entrada) e 404 (quando um parâmetro não existe, por exemplo).

De seguida, abordaremos com mais detalhe cada um dos endpoints, mencionando como os construímos, quais os comandos SQL utilizados e as validações efetuadas e o landing page.

Landing Page

Para esta etapa decidimos criar um logo para a nossa Pet Store fictícia. Criamos uma função com o nome *landing_page()*, onde definimos uma pequena frase de boas vindas e colocamos o url para o nosso logo ser mostrado, assim como a abreviatura da cadeira e o ano letivo.

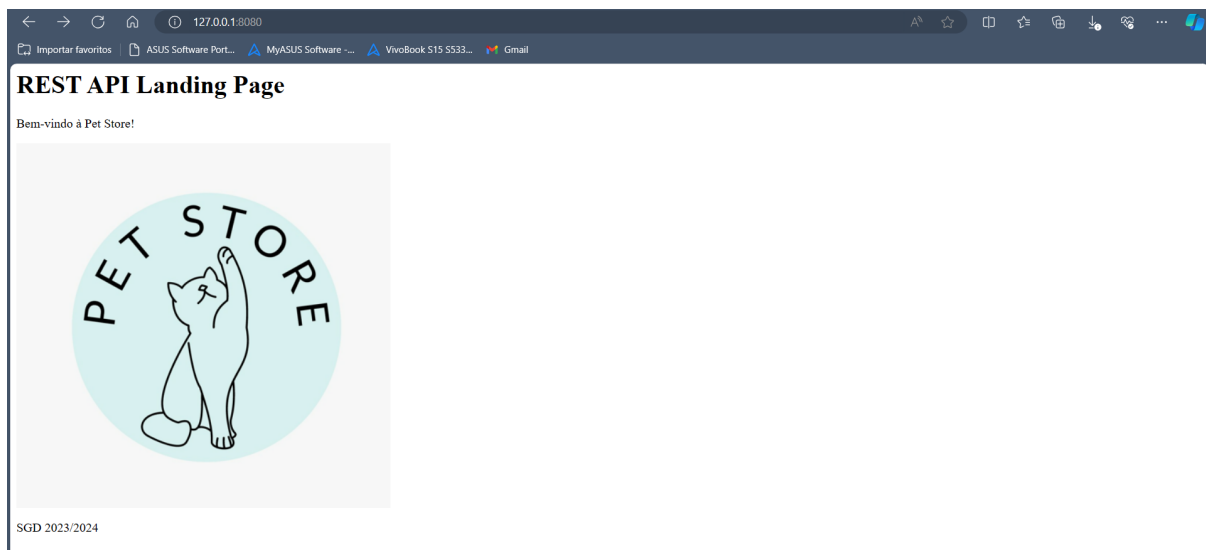


Fig.6 - Landing Page

Endpoint 1: Criar um Item

O primeiro endpoint, conforme mencionado na tabela, está disponível através do método POST em <http://localhost:8080/proj/api/items> e destina-se à criação de novos itens no sistema.

O utilizador deve fornecer informações sobre o item que pretende criar, como nome, categoria, preço, quantidade em stock, descrição, fabricante, peso e URL da imagem. Foram aplicadas várias validações: verificamos os parâmetros para garantir que o utilizador nos fornece todas as informações necessárias na requisição; verificamos os valores para assegurar que tanto o preço como o stock e o peso sejam positivos; e verificamos as categorias existentes, se o item adicionado tiver uma categoria inexistente na loja, o utilizador terá a opção de criar ou não a categoria. Se optar por não criar, receberá uma mensagem de erro correspondente à criação do item.

Em seguida, com o comando INSERT da linguagem SQL, adicionamos estas informações à base de dados e criamos um item com estas características. O índice associado é *AUTOINCREMENT* (AU) portanto a cada criação incrementa sozinho.

A resposta é JSON e indica o sucesso da operação, uma mensagem a avisar que o item foi criado e detalhes sobre este, tais como o seu ID, nome, categoria, preço, quantidade em stock, descrição, fabricante, peso, URL da imagem e o número total de unidades vendidas, que inicialmente será 0 visto que o produto acabou de ser criado.

Endpoint 2: Atualizar um Item

O segundo endpoint atualiza um item específico e está acessível por meio do método PUT. A URL é adaptada conforme o produto que desejamos atualizar. Por exemplo, um item com o índice 1234, acessamos a <http://localhost:8080/proj/api/items/1234> onde os últimos dígitos correspondem a esse ID. Este endpoint permite a modificação de diversos atributos incluindo nome, categoria, preço, stock, descrição, fabricante, peso e URL da imagem.

São realizadas uma série de validações, como a verificação da existência do item pelo seu ID, a verificação e criação de uma nova categoria (estratégia análoga à do endpoint anterior), validação dos parâmetros de atualização para garantir que pelo menos um seja válido e validação de valores para assegurar que o preço, o stock e o peso sejam positivos.

Depois de passar por todos estes testes de validação, o sistema atualiza a base de dados com as novas informações com o uso do comando UPDATE do SQL. Finalmente, retornará uma resposta JSON, indicando o sucesso da operação, uma mensagem de confirmação que o item foi atualizado e os detalhes sobre ele como o seu ID (que não é alterável), nome, categoria, preço, stock, descrição, fabricante, peso e URL de imagem.

Endpoint 3: Remover um Item do Carrinho

Este endpoint é responsável por remover um item específico do carrinho de um determinado cliente. A URL é adaptada consoante o cliente e o ID do item que estão em questão: http://localhost:8080/proj/api/carts/{client_id}/items/{item_id} e utiliza o método DELETE.

São efetuadas verificações para confirmar a existência do cliente e do item e a presença do item no carrinho deste cliente. Após isto, o produto será removido através do comando DELETE. Desta forma, a resposta JSON indicará o sucesso da operação com uma mensagem de confirmação que o item foi removido.

Endpoint 4: Adicionar um Item ao Carrinho

O quarto endpoint está disponível através do método POST e é adaptável conforme o cliente dono do carrinho em http://localhost:8080/proj/api/cart/{client_id}. Este endpoint possibilita a adição de um item ao carrinho de compras de um cliente específico. Enquanto o

ID do cliente é fornecido no URL, as informações do item, como o seu ID e a quantidade desejada a adicionar, são enviadas na requisição.

Durante a execução, são verificadas a existência do carrinho, e os parâmetros da requisição, que devem incluir tanto o ID do item como a quantidade (a qual deve ser positiva, como também é confirmado). Se todas as verificações forem bem-sucedidas, o item é inserido conforme desejado através do comando INSERT do SQL e é retornada uma resposta JSON que menciona o sucesso acompanhada por uma mensagem de confirmação da inserção.

Endpoint 5: Obter uma Lista de Itens

No contexto deste endpoint, é utilizado o método GET para a funcionalidade de mostrar uma lista de itens com determinadas características, todas opcionais. Estas incluem o número da página à qual queremos aceder, quantos itens cada página tem, o filtro por categoria e ordenação dos itens ou por nome ou por preço. A estrutura da URL segue uma lógica simples, com o início padrão em <http://localhost:8080/proj/api/items>, e para cada caso específico, adicionamos os parâmetros opcionais:

- Para especificar a página desejada, utilizamos [page={num_pagina}](#), sendo 1 o valor padrão.
- Para definir o número de itens por página, que por padrão é 10, usamos [pageSize={quantidade_por_pagina}](#).
- Para ordenar, usamos [sort=price](#) ou [sort=name](#).
- Para filtrar por categoria, usamos [category={categoria}](#).

Alguns exemplos de uso incluem:

- <http://localhost:8080/proj/api/items?page=1&pageSize=10>: para aceder à primeira página de 10 itens.
- <http://localhost:8080/proj/api/items?sort=price>: para ordenar os itens por preço.
- <http://localhost:8080/proj/api/items?sort=name&page=2&pageSize=7>: para ordenar por nome e selecionar a segunda página com 7 itens.

Neste endpoint, verificamos a requisição para garantir que os parâmetros sejam válidos, isto é, o número de páginas e a quantidade de itens devem ser inteiros positivos, a existência da categoria pedida na base de dados é analisada, e se o parâmetro de ordenação corresponde ao nome ou ao preço, pois não corresponder ou se estiver mal escrito, retorna o devido erro.

Se tudo estiver correto, é retornada uma resposta JSON que indica sucesso, uma mensagem que transmite o sucesso, e a página com as características pedidas.

Endpoint 6: Obter os detalhes de um item

Neste endpoint o objetivo é obter os detalhes de um item selecionado. A URL pode ser adaptada consoante o item ao qual pretendemos visualizar:

<http://localhost:8080/proj/api/items/{id}> e utiliza o método GET.

É feita uma pesquisa pelo id do item que queremos obter os detalhes na tabela “item” através do comando SELECT e depois, caso o item tenha sido encontrado, devolvemos a

resposta que corresponde às informações presentes em relação a esse item, caso contrário, o código devolve um erro “Item not found”.

Endpoint 7: Pesquisar itens

O endpoint 7 corresponde a uma pesquisa de itens pelo nome. A URL tem na sua composição o nome pelo qual pretendemos pesquisar:

<http://localhost:8080/proj/api/items/search/{search}> e utiliza o método GET.

Começamos por realizar uma pesquisa do nome do item através do comando SELECT na tabela *item* e caso a pesquisa tenha sido bem sucedida, a resposta é os detalhes dos itens correspondentes à pesquisa, caso contrário a resposta será um erro “No itens found for the given search criteria.”.

Endpoint 8: Obter o Top 3 de Vendas por Categoria

Para a execução deste endpoint, que tem como objetivo retornar os itens mais vendidos em cada categoria com base na quantidade total de vendas, utilizamos o método GET em <http://localhost:8080/proj/api/stats/sales>.

Utilizamos o comando SELECT numa tabela construída através da junção das tabelas *purchaseitem*, *purchase* e *category* com o comando JOIN, e conseguimos obter os dados relativos à quantidade total de vendas para cada item em cada categoria. Agrupamos por categoria com GROUP e ordenamos a quantidade total de vendas por ordem decrescente com ORDER. Desta forma, conseguimos identificar os três itens mais vendidos em cada categoria.

Endpoint 9: Compra de Itens

Este endpoint consiste na compra de itens que estejam no carrinho de compras de um determinado cliente. A URL é <http://localhost:8080/proj/api/purchase> e o método utilizado foi o POST.

O utilizador deve fornecer informações como a sua identificação, os itens que deseja comprar e a respetiva quantidade.

Utilizamos SELECT para selecionar um item do carrinho de compras onde esteja o ID do cliente, caso encontre usamos outro SELECT para encontrar o stock e o preço do item e depois calculamos um novo valor de stock subtraindo a quantidade de produto que vai ser comprada e fazemos UPDATE do stock do item. Depois realizamos um INSERT na tabela *purchase* dos dados necessários, o preço total, a data da compra e o ID do cliente, realizando assim a compra pretendida.

Para validar os dados, verificamos se o carrinho está presente para o cliente especificado, se os itens pedidos estão no carrinho e se o stock era suficiente para a quantidade pedida.

Este endpoint continha problemas de controle de transações que tivemos de controlar. Inicialmente, para iniciar manualmente a transação, definimos *conn.autocommit* como False, o que significa que as operações subsequentes não serão efetivas automaticamente. Logo de seguida, utilizamos *conn.begin()* para iniciar a transação, de forma a garantir mais clareza e controle sobre as operações realizadas. Após todas serem realizadas com sucesso, chamamos *conn.commit()* que irá confirmar a transação. Isto tornará efetivas todas as alterações que até então tinham sido efetuadas no banco de dados. No caso de ocorrer um erro durante o processo, assim como feito noutros endpoints, chamamos *conn.rollback()* para reverter as alterações. Desta forma evitamos a concorrência e prevenimos que o mesmo item seja comprado duas vezes.

Endpoint 10: Obter Clientes com Filtros

O décimo endpoint fornece informações sobre os clientes com base em filtros específicos, que incluem a data da última compra e o último item comprado. Para executar esta operação, utilizamos o método GET em <http://localhost:8080/proj/api/client>.

Durante a execução, demos uso aos comandos SELECT e LEFT JOIN de forma a juntar as tabelas *client*, *purchase*, *purchaseitem* e *item*. Utilizamos WHERE para filtrar os resultados conforme os critérios fornecidos, ou seja, para encontrar a data da última compra e o último item comprado, isto somente se estes filtros existirem, tal como temos o cuidado de o verificar.

Assim, se nenhum erro foi encontrado, surge uma resposta JSON que indica o sucesso da operação, junta com todos os detalhes de cada cliente como o seu ID, nome, email, a data da última compra e o último item comprado. Se estes últimos 2 forem null, significa que o cliente ainda não efetuou nenhuma compra.

Endpoint 11: Adicionar Cliente

No seguinte endpoint o objetivo será adicionar um cliente. Utilizamos o método POST no seguinte URL <http://localhost:8080/proj/api/clients>.

Começamos por contar todos os clientes que há na tabela *client* e inicializar o ID do cliente nessa conta mais 1. Depois fazemos um INSERT dos valores id do cliente, nome e email na tabela *client* inserindo assim um novo cliente.

Endpoint 12: Obter Compras de Clientes

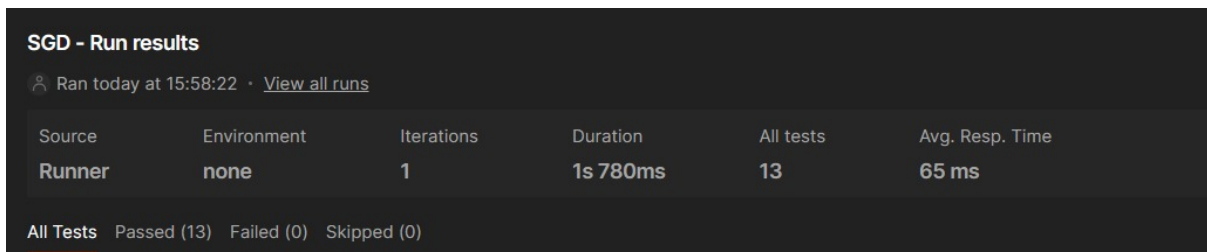
Neste endpoint, pretendíamos obter detalhes sobre as compras de um cliente específico, como ID do pedido, preço total, data do pedido e os itens comprados. Para isto, foi utilizado o método GET na URL http://localhost:8080/proj/api/clients/{client_id}/orders que varia conforme o cliente que desejamos analisar.

Selecioneamos os dados que queriamos obter com SELECT, utilizamos JOIN para agrupar as tabelas *purchase* e *purchaseitem* e selecionamos todas as amostras que continham este determinado cliente com WHERE.

Como validação, verificamos a existência do cliente no banco de dados. Se nenhum erro foi retornado, a função gera uma resposta JSON que indica o sucesso da operação. De seguida, apresenta as informações desejadas de todos os pedidos que o cliente já fez.

Testes e Erros

Para testarmos os nossos endpoints utilizamos o *Postman*. O ficheiro relativo aos testes encontra-se no zip com o nome *postman*. Durante o desenvolvimento do nosso projeto fizemos inúmeros testes para cada endpoint, no entanto no ficheiro colocamos apenas 1 para cada, só para exemplificar, com exceção do 5 em que colocamos 2 exemplos. Como teste utilizamos um que no caso de o código de resposta da API for 200, considera o teste bem sucedido e caso seja diferente considera que falhou. Os resultados obtidos para os 12 endpoints foram positivos, como podemos observar pela figura 7.



Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	1s 780ms	13	65 ms

All Tests Passed (13) Failed (0) Skipped (0)

Fig.7 - Resultados dos Testes no Postman

Durante todo o processo de criação dos endpoints, tivemos o cuidado de tratar e identificar os erros que podiam ocorrer. As figuras abaixo vão mostrar alguns exemplos.

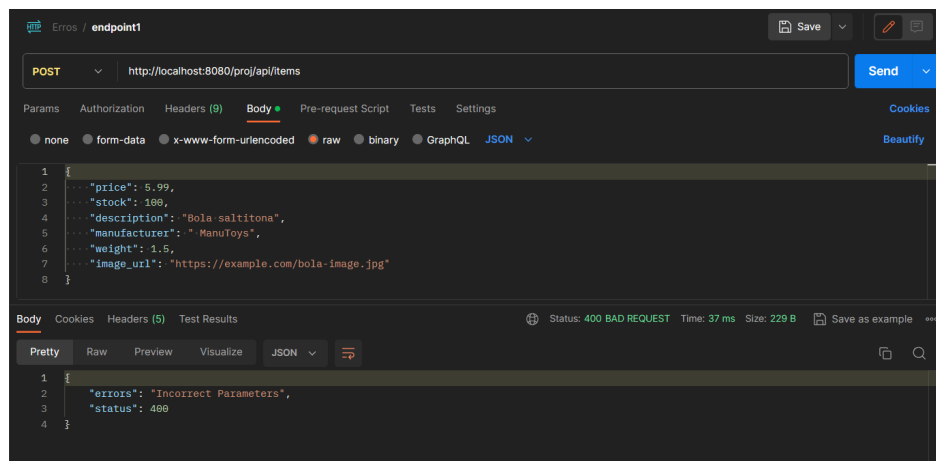


Fig. 8 - Erro quando há parâmetros em falta (exemplo para *endpoint1*)

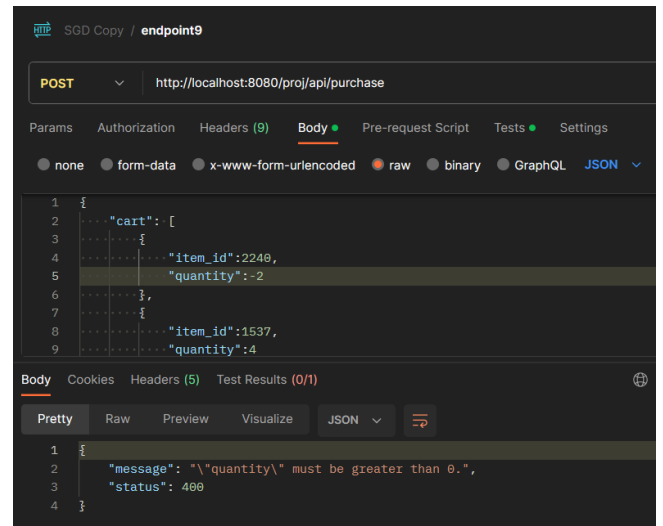
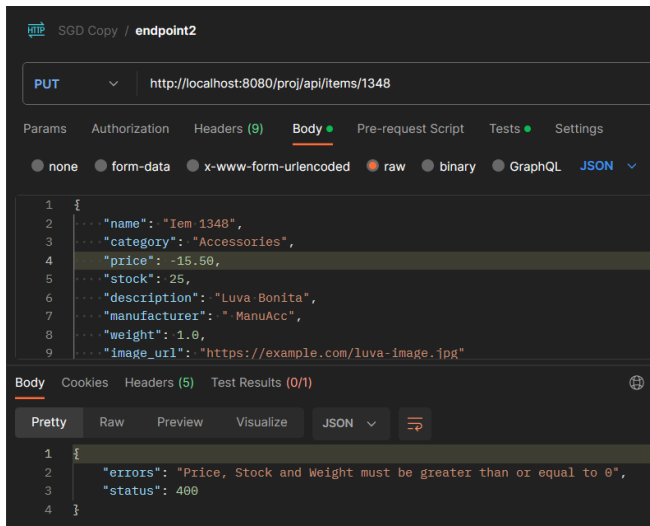


Fig. 9 - Erro quando há certos parâmetros negativos (exemplo *endpoint2* e *endpoint9*)

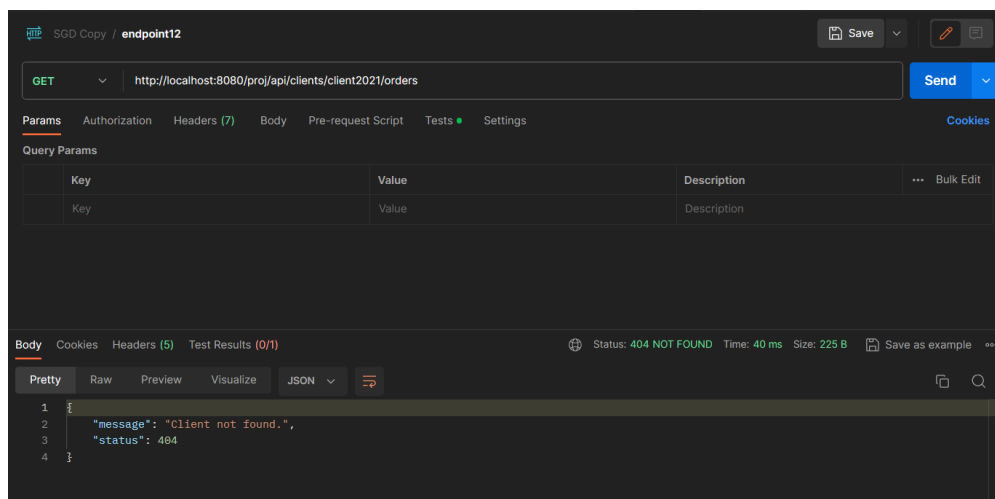


Fig. 10 - Erro quando o índice do cliente não é válido (exemplo *endpoint12*)

É importante referir novamente que estes são apenas alguns dos erros que tivemos em consideração durante as nossas validações. Quando os processos são executados sem erros aparece uma mensagem a dizer que foi bem sucedido e o status 200. Podemos ver um exemplo na figura 11.

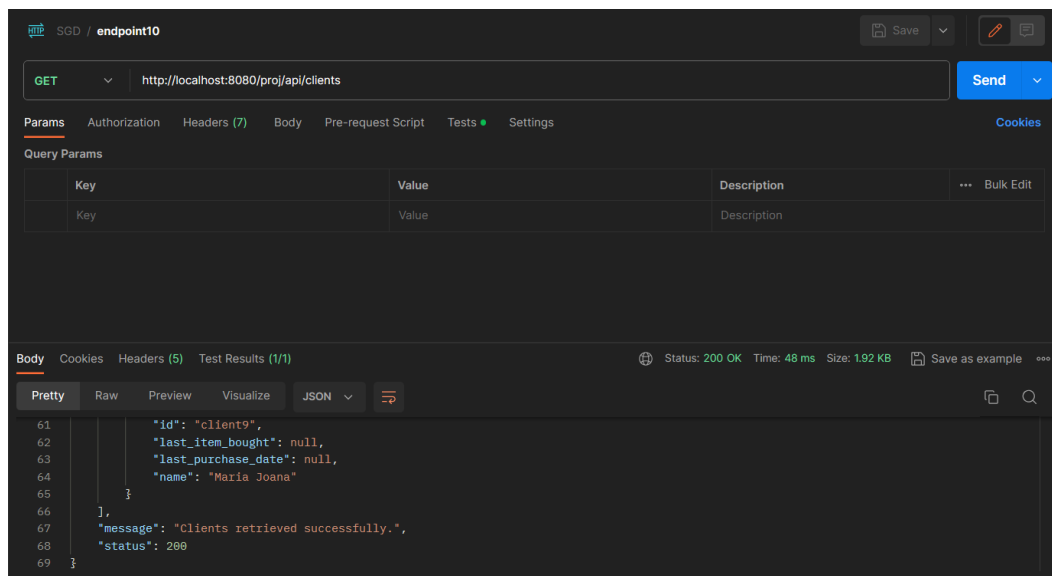


Fig.11 - Output quando o processo é realizado com sucesso (exemplo *endpoint10*)

Gráfico de Esforço dos Elementos

