

**UNIFACS - UNIVERSIDADE SALVADOR**  
**UC – SISTEMAS DISTRIBUÍDOS E MOBILE**

Douglas Dórea da Silva Melo - RA: 1272215019

Marcos Vinicius de Jesus Galvão - RA: 12722112437

Ricardo Brito da Silva - RA: 12722129482

Wilson Pinheiro dos Santos Neto - RA: 12723110270

**SIMULAÇÃO DA CAPTAÇÃO DE DADOS DE VENDA DE UMA REDE DE LOJAS:**  
**PLATAFORMA DOMINUS PRO**

**SALVADOR**

**2023**

Douglas Dórea da Silva Melo - RA: 1272215019

Marcos Vinicius de Jesus Galvão - RA: 12722112437

Ricardo Brito da Silva - RA: 12722129482

Wilson Pinheiro dos Santos Neto - RA: 12723110270

SIMULAÇÃO DA CAPTAÇÃO DE DADOS DE VENDA DE UMA REDE DE LOJAS:

PLATAFORMA DOMINUS PRO

Este relatório tem como finalidade apresentar uma visão macro do processo de desenvolvimento da plataforma Dominus Pro.

Orientador: Eduardo Sidney da Silva Xavier.

SALVADOR

2023

## Como executar a aplicação:

Este projeto é dividido em duas partes:

1. Backend (Servidor);
2. Frontend (Cliente);

**O front-end precisa que o back-end esteja sendo executado para funcionar.**

### Pré-requisitos:

Antes de começar, você vai precisar ter instalado em sua máquina as seguintes ferramentas: [Git](#), [Node.js](#), [MYSQL](#), Além disto é bom ter um editor para trabalhar com o código como [Visual Studio Code](#) e uma ferramenta para banco de dados como [DBEAVER](#).

### Rodando back-end com docker-compose:

Para iniciar a aplicação com Docker, você vai precisar clonar o repositório utilizando o [Git](#), é bom ter um editor para trabalhar com o código como [Visual Studio Code](#) e uma ferramenta para banco de dados como [DBEAVER](#), além disso, o [docker](#) (essa instalação serve para o linux) e [docker-compose](#) (essa instalação serve para o linux) também. Basicamente você vai acessar a pasta raiz do back-end no repositório, com os pré-requisitos instalados, vai criar um arquivo “.env” e copiar o conteúdo do “.env-example-docker” e colar dentro do “.env”, depois disso vai abrir o terminal na raiz do back-end e rodar o comando no terminal “docker-compose up -d --build”.

Observação: Também foi adicionado no back-end a opção de rodar com docker (opcional) se já tiver conhecimento e quiser realizar por ser mais prático.

Rodando o backend (servidor):

Certifique-se de ter a versão **v18.17.1** do Node instalada, certifique-se de já possuir os pré-requisitos instalados.

Clone esse repositório: `git clone` **REPOSITÓRIO**;

Acesse a pasta do projeto que acabou de clonar;

Instale as dependências utilizando: `npm install` ou `yarn install` (varia de acordo com o gerenciador de pacotes instalado na máquina do usuário);

Crie um arquivo chamado `“.env”` na raiz do projeto;

Procure o arquivo `“.env.example”` e copie os dados do arquivo e cole no `“.env”`

Altere os dados do `“.env”` e mude para os dados do seu ambiente;

`APP_PORT=` insira a porta que deseja rodar o servidor, geralmente usa-se a `"3000"`.

`DATABASE_HOST=` aqui geralmente usa-se `"localhost"`.

`DATABASE_NAME=` insira o nome do banco de dados que você quer criar, ele vai gerar o banco pelo nome dessa variável.

`DATABASE_USERNAME=` insira o usuário do banco de dados, geralmente é `"root"`.

`DATABASE_PASSWORD=` insira a senha do seu banco de dados.

`DATABASE_PORT=` insira a porta, geralmente é: `"3306"`.

`DATABASE_SYNCHRONIZE=` insira `true` na primeira vez que for rodar o projeto para criar o banco e a tabela e depois coloque como `false`.

Execute a aplicação em modo de desenvolvimento

`npm run dev` **OU** `yarn dev`

# O servidor iniciará na porta:4000 por padrão - acesse `http://localhost:4000`

# Observações:

# Essa porta é referente ao `APP_PORT` presente no `“.env”`, se você alterar, terá que mudar a porta `"4000"` para a que você definiu.

Rodando o frontend (cliente):

Certifique-se de ter a versão **v18.17.1** do Node instalada, certifique-se de já possuir os pré-requisitos instalados.

Clone esse repositório: `git clone REPOSITÓRIO`;

Acesse a pasta do projeto que acabou de clonar;

Instale as dependências utilizando: `npm install` ou `yarn install` (varia de acordo com o gerenciador de pacotes instalado na máquina do usuário);

Crie um arquivo chamado `“.env”` na raiz do projeto;

Procure o arquivo `“.env.example”` e copie os dados do arquivo e cole no `“.env”`

Altere os dados do `“.env”` e mude para os dados do seu ambiente;

`VITE_BASE_URL=` insira a porta que deseja rodar o servidor, geralmente usa-se a `"3000"`.

Altere a porta ex: `http://localhost:SUAPORTADOBACK`

Execute a aplicação em modo de desenvolvimento

`npm run dev` **OU** `yarn dev`

O cliente iniciará na porta:5173 por padrão - acesse `http://localhost:5173`

Observação: Essa aplicação só vai funcionar perfeitamente com o backend em node rodando em paralelo, ou seja, ao mesmo tempo.

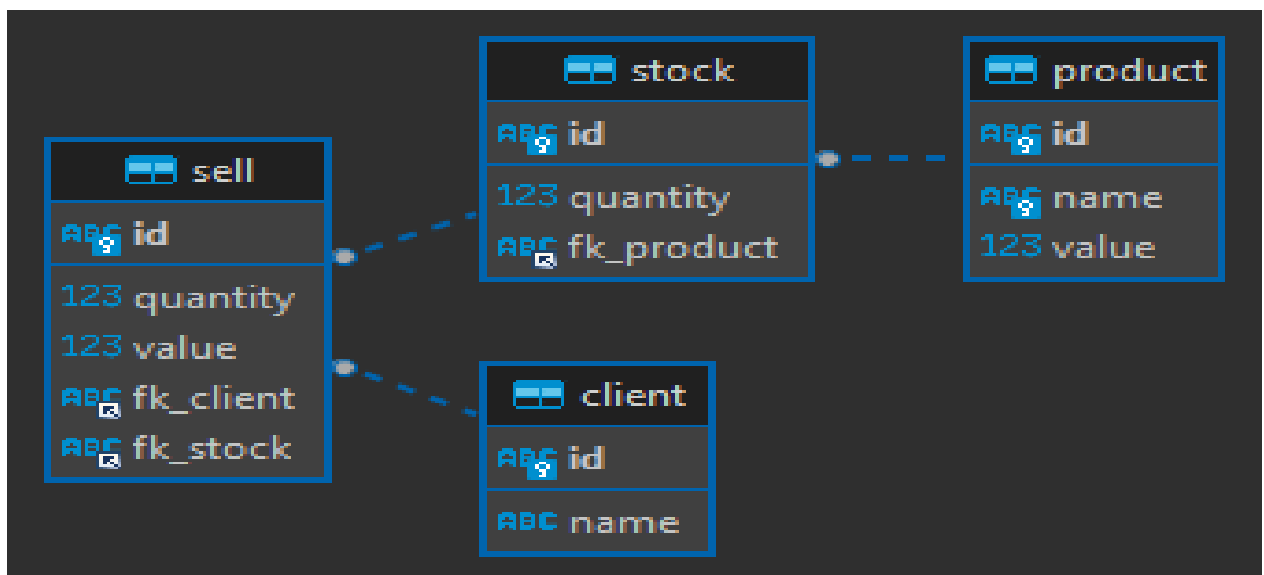
## Arquitetura, estratégia e algoritmos utilizados:

### Banco de Dados

Para o desenvolvimento da aplicação, utilizamos o banco MYSQL pela facilidade de integração com tecnologias existentes e pela velocidade e desempenho.

### Modelo Entidade Relacionamento

A aplicação foi desenvolvida seguindo a ideia de cadastrar um produto (product) por venda (sell) para um cliente (client) específico, considerando os requisitos propostos, também temos um estoque (stock) que pode armazenar uma quantidade de um produto.



## Servidor (API):

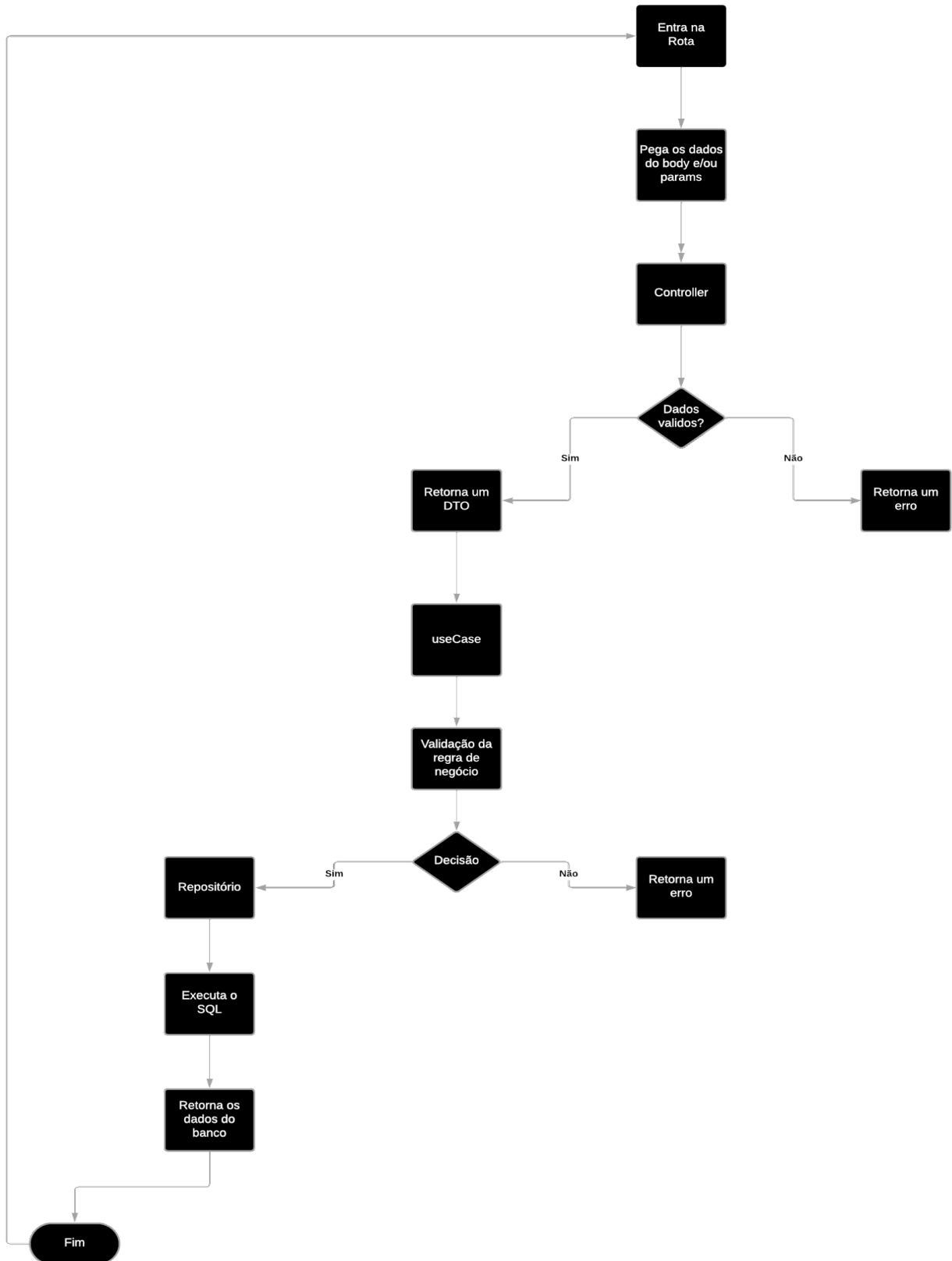
Para atuar no lado do servidor, optamos pela criação de uma API (Application Programming Interface). O que fornece alguns benefícios como: flexibilidade de plataforma, padrão de comunicação, facilidade de integração, escalabilidade e manutenção.

A aplicação foi desenvolvida em Node.JS utilizando uma arquitetura em camadas, essa abordagem foi escolhida para manter o código com uma alta manutenibilidade e com o objetivo de separar o máximo de responsabilidade possível. Nesse sentido também foi utilizado alguns elementos de designs famosos da programação como: Domain Driven Design e Clean Architecture.

A estrutura final do projeto e suas responsabilidades foram divididas da seguinte forma:

- controller: Responsável por receber inputs e validar os dados para serem enviados ao caso de uso;
- dto: Ele vai devolver um objeto com os atributos (chave, valor) necessários para o caso de uso;
- framework-drivers:
  - database: Lida com a configuração e interação com o banco de dados;
  - libraries: Armazena bibliotecas externas usadas no projeto;
- routes: Gerencia as rotas (endpoints) da aplicação, criando requisições (POST, GET, PUT, DELETE) e mostrando o fluxo o qual deve ser seguido.
- use-cases: Armazena os casos de uso da aplicação. Cada caso de uso é uma interação específica do usuário, encapsulando a regra de negócio;
- util: Contém funções que fornecem funcionalidades comuns reutilizáveis em toda aplicação.
- server.js: arquivo que funciona como ponto de entrada da aplicação, onde é configurado o servidor e conecta todas as partes.

## Fluxo da API:





Rota:

A rota tem como papel fazer o direcionamento de requisições HTTP para funções específicas com base na URL e métodos HTTP.

```
1  clientRoutes.post("/client", async (request, response) => {
2    try {
3      const input = request.body;
4
5      const validCreateClientDto = createClientController(input);
6
7      if (!validCreateClientDto) {
8        throw {
9          message: "Os dados de entrada não são válidos.",
10         status: HttpStatus.UNPROCESSABLE_ENTITY,
11       };
12     }
13
14     const output = await createClientUseCase(validCreateClientDto);
15
16     response.status(HttpStatus.CREATED).json(output);
17   } catch (error) {
18     response.status(error.status).json({
19       error: error.message,
20     });
21   }
22 });
```


Controller:

A função do controller dentro dessa aplicação é para validação de dados, cada caso de uso, vai ter um controller específico que vai fazer uma validação dos dados de entrada e pode ou retornar um erro ou um DTO.


```
1  function createClientController(input) {
2    if (Object.keys(input).length < 0) {
3      throw {
4        message: "Não foi recebido nenhum campo.",
5        status: HttpStatus.UNPROCESSABLE_ENTITY,
6      };
7    } else if (input.name.length == 0) {
8      throw {
9        message: "O campo nome não pode ser vazio.",
10       status: HttpStatus.UNPROCESSABLE_ENTITY,
11     };
12   } else if (input.name.length > 14) {
13     throw {
14       message: "O campo nome não pode ter mais que 14 caracteres.",
15       status: HttpStatus.UNPROCESSABLE_ENTITY,
16     };
17   } else if (typeof input.name != "string") {
18     throw {
19       message: "O campo nome deve ser do tipo string.",
20       status: HttpStatus.UNPROCESSABLE_ENTITY,
21     };
22   }
23
24   return inputClientDto(input);
25 }
```

DTO (Data Transfer Object):

O DTO é um objeto de transferência de dados, nessa aplicação, ele vai receber os dados no controller caso não tenha nenhum problema e vai retornar um objeto com os dados necessários para um caso de uso em específico.



```
1  function inputClientDto(input) {  
2    return {  
3      id: crypto.randomUUID(),  
4      name: input.name,  
5    };  
6  }
```



```
1  function outputClientDto(input) {  
2    return {  
3      id: input.id,  
4      name: input.name,  
5    };  
6  }
```

Use Case (Caso de Uso):

O caso de uso vai receber os dados (DTO) da rota após todas as validações e vai ser a camada que vai ter comunicação com o repositório (que interage com o banco de dados), ele também vai ser responsável por validar regras de negócios da aplicação e aplicar a lógica dos casos de uso.

```
1  async function createClientUseCase(client) {
2    try {
3      const alreadyExists = await getClientByNameRepository(client.name);
4
5      if (alreadyExists.length > 0) {
6        const error = new Error("Já existe um cliente com esse nome.");
7        error.status = HttpStatus.BAD_REQUEST;
8        throw error;
9      }
10
11     const newClient = await createClientRepository(client);
12
13     return outputClientDto(newClient);
14   } catch (error) {
15     const status = error.status || HttpStatus.INTERNAL_SERVER_ERROR;
16     throw {
17       message: `Erro ao criar um cliente: ${error.message}`,
18       status: status,
19     };
20   }
21 }
```

## Repository (Repositório):

O repositório nesse contexto é a camada de comunicação direta com o banco de dados, nessa camada que vai ter a consulta SQL e vai acontecer a comunicação com o banco de dados. Essa camada vai ser responsável por desempenhar o papel de interagir com o banco de dados e realizar alguma operação do CRUD (Create, Read, Update, Delete).

```
1  async function createClientRepository(client) {
2    try {
3      await database.query("INSERT INTO client (id, name) VALUES (?, ?)", [
4        client.id,
5        client.name,
6      ]);
7
8      const createdUser = await database.query(
9        `SELECT * FROM client WHERE id = ?`,
10       [client.id]
11     );
12
13     return createdUser.length > 0 ? createdUser[0] : [];
14   } catch (error) {
15     throw error;
16   }
17 }
```

Lógica para geração dos relatórios:

Para a criação dos relatórios, optamos pela exportação via xlsx (excel) do relatório com todas as informações, para realizar essa exportação utilizamos uma biblioteca chamada de excel-4-node (<https://www.npmjs.com/package/excel4node>).

A biblioteca já fornece as funções necessárias para criar o xlsx, dessa forma, foi criado um arquivo “excel-4-node” (src/framework-drivers/libraries) e dentro dele uma única função, essa função vai receber por parâmetro um “data” (esse ‘data’ vai ser os dados recebidos do caso de uso), vai receber um parâmetro “headingColumnNames” que vai ser um array com as colunas que vão ser preenchidas no xlsx, e o nome que é o nome do arquivo final.

Já dentro da função, ele vai percorrer o array com as colunas que vão ser preenchidas no xlsx e adicionar no arquivo.

Depois ele vai percorrer os dados (data) e para cada objeto do array que ele percorrer, ele vai iterar sobre todas as chaves do objeto record. No segundo forEach, dentro desse segundo forEach, ele vai acessar a célula na planilha na posição ‘rowIndex, columnIndex’ e inserir o valor da propriedade correspondente ao nome da coluna atual ‘columnName’ no objeto ‘record’. Por fim, após cada objeto record ser completamente percorrido, ele vai incrementar o rowIndex e incrementar para mover para a próxima linha na planilha.

No final, ele vai gerar uma promise que recebe duas funções uma para indicar o sucesso (resolve) e outra a falha (reject), a promise é utilizada para indicar uma função assíncrona.

Ele nomear o arquivo, após isso, vai ‘escrever o arquivo’ (vai para a raiz do projeto) e caso tenha sucesso, ele vai ler o arquivo gerado e transformar em base64 para ser enviado como resposta da requisição.

A lógica para geração do arquivo e transformar em base64, e vai ser a mesma para todos os relatórios, a única coisa que vai mudar são os conteúdos dos parâmetros que vão ser passados para função.

```
1  async function generateXlsx(data, headingColumnNames, name) {
2    const wb = new xl.Workbook();
3    const ws = wb.addWorksheet("Worksheet Name");
4
5    let headingColumnIndex = 1;
6    headingColumnNames.forEach((heading) => {
7      ws.cell(1, headingColumnIndex++).string(heading);
8    });
9
10   let rowIndex = 2;
11   data.forEach((record) => {
12     let columnIndex = 1;
13     Object.keys(record).forEach((columnName) => {
14       ws.cell(rowIndex, columnIndex++).string(record[columnName].toString());
15     });
16     rowIndex++;
17   });
18
19   return new Promise((resolve, reject) => {
20     const filePath = `${name}.xlsx`;
21
22     wb.write(filePath, (error, stats) => {
23       if (error) {
24         reject(error);
25       } else {
26         const fileContent = fs.readFileSync(filePath, { encoding: "base64" });
27         resolve({ fileContent });
28       }
29     });
30   });
31 }
```

Product Lower Stock (Produto com baixo estoque):

Esse relatório vai exportar os campos: 'nome', 'valor', 'quantidade' dos produtos com baixo estoque, a regra de negócio implementada para o produto ser considerado como “baixo estoque” é que ele tenha em estoque, uma quantidade inferior ou igual a 5.

O useCase onde recebe de fato a lógica do negócio da aplicação vai receber em ordem uma lista (array) dos produtos em estoque do repositório, logo em seguida, vai verificar se realmente tem dados nessa lista, se não tiver vai gerar uma exceção e encerrar o caso de uso, caso contrário, ele vai iterar pela lista transformando cada dado em um outputStockDto (que vai formatar e fazer as devidas conversões dos dados), depois ele vai filtrar nesse array os produtos no estoque que tem uma quantidade menor ou igual a 5. Vai iterar novamente no objeto já filtrados e reorganizar de forma crescente. Vai gerar um novo array, extraíndo os campos relevantes e que vão ser utilizados na geração do relatório e retornar esse array.





```
1  async function productLowerStockUseCase() {
2    try {
3      const stock = await listStockRepository();
4
5      if (stock.length === 0) {
6        const error = new Error(
7          "Não é possível listar os produtos com baixo estoque, pois ainda não foi tem nenhum produto no estoque com menos de 10 unidades."
8        );
9        error.status = HttpStatus.BAD_REQUEST;
10       throw error;
11     }
12
13     const stockDto = stock.map((item) => outputStockDto(item));
14
15     const lowerStockProducts = stockDto.filter((stock) => stock.quantity <= 5);
16
17     const lowerStockProductsOrder = Object.values(lowerStockProducts).sort(
18       (a, b) => a.quantity - b.quantity
19     );
20
21     const extractFields = lowerStockProductsOrder.map((stockProduct) => ({
22       nome: stockProduct.product_name,
23       valor: stockProduct.product_value,
24       quantidade: stockProduct.quantity,
25     }));
26
27     return extractFields;
28   } catch (error) {
29     const status = error.status || HttpStatus.INTERNAL_SERVER_ERROR;
30     throw {
31       message: `Erro ao listar os produtos mais vendidos: ${error.message}`,
32       status: status,
33     };
34   }
35 }
```

## Products Per Client (Produtos por cliente):

O useCase onde recebe de fato a lógica do negócio da aplicação vai receber em ordem uma lista (array) dos produtos vendidos por um cliente em específico vindo do repositório, logo em seguida, vai verificar se realmente tem dados nessa lista, se não tiver vai gerar uma falha e encerrar o caso de uso, caso contrário, ele vai iterar pela lista transformando cada dado em um `outputSellDto` (que vai formatar e fazer as devidas conversões dos dados), depois ele vai iterar pelo array agrupando os produtos vendidos por cliente e calculando estatísticas de venda para cada produto, como número de pedidos, total de vendas e quantidade de produtos vendidos.

Ao iterar no array, utilizando o `reduce`, ele vai verificar se um cliente já foi registrado no objeto `“bestSellingProducts”` e caso não tenha, ele vai criar uma entrada no objeto para esse cliente. Para cada cliente, ele vai verificar se um determinado produto da lista já foi vendido anteriormente, se o produto não tiver na lista de produtos vendidos para o cliente, ele vai adicionar. A cada passagem pelo objeto, ele vai contabilizar a quantidade de pedidos, o total de vendas e as informações do produto incrementando a cada produto igual.

Por fim, vai retornar o objeto `“bestSellingProducts”` que contém os produtos vendidos por cada cliente, juntamente com estatísticas calculadas para cada produto (total de pedidos, valor total de vendas e quantidade total vendida para o cliente).

```

1  async function getProductsPerClientUseCase(input) {
2    try {
3      const orders = await getSellByClientRepository(input.fk_client);
4
5      if (orders.length === 0) {
6        const error = new Error(
7          "Não é possível listar os produtos desse cliente, pois ainda não foi vendido nenhum produto pra esse cliente."
8        );
9        error.status = HttpStatus.BAD_REQUEST;
10       throw error;
11     }
12
13     const sellsDto = orders.map((item) => outputSellDto(item));
14
15     const bestSellingProducts = sellsDto.reduce((accumulator, currentValue) => {
16       const clientName = currentValue.client_name;
17       const productName = currentValue.product_name;
18
19       if (!accumulator[clientName]) {
20         accumulator[clientName] = [];
21       }
22
23       const existingProduct = accumulator[clientName].find(
24         (product) => product.product_name === productName
25       );
26
27       if (!existingProduct) {
28         accumulator[clientName].push({
29           orderRequests: 1,
30           totalSale: currentValue.totalOrder,
31           product_name: productName,
32           product_value: currentValue.product_value,
33           orderProductQuantity: currentValue.orderProductQuantity,
34         });
35       } else {
36         existingProduct.orderRequests += 1;
37         existingProduct.totalSale += currentValue.totalOrder;
38         existingProduct.orderProductQuantity +=
39           currentValue.orderProductQuantity;
40       }
41
42       return accumulator;
43     }, {});
44
45     const bestSellingProductsArray = Object.values(bestSellingProducts).flat();
46
47     const orderBestSellingProducts = bestSellingProductsArray.sort(
48       (a, b) => b.orderRequests - a.orderRequests
49     );
50
51     const extractFields = orderBestSellingProducts.map(
52       (bestSellingProduct) => ({
53         "Nome do produto": bestSellingProduct.product_name,
54         "Valor por produto": bestSellingProduct.product_value,
55         "Unidades vendidas": bestSellingProduct.orderProductQuantity,
56         "Quantidade de vendas": bestSellingProduct.orderRequests,
57         "Valor total": bestSellingProduct.totalSale.toFixed(2),
58       })
59     );
60
61     return extractFields;
62   } catch (error) {
63     const status = error.status || HttpStatus.INTERNAL_SERVER_ERROR;
64     throw {
65       message: `Erro ao listar os produtos por cliente: ${error.message}`,
66       status: status,
67     };
68   }
69 }
70

```

Average client consumption (Média de consumo por cliente):

Esse caso de uso, utiliza uma biblioteca para gestão de datas chamada moment.js (<https://momentjs.com/>), o uso dessa biblioteca é limitado nesse contexto, a pegar a quantidade de dias do mês vigente. O caso de uso começa recebendo a quantidade de vendas de um cliente específico, verificando se existem dados disponíveis. Caso não tenha nenhuma venda para o cliente, ele lança uma exceção indicando a impossibilidade no cálculo.

Após recuperar os dados do repositório (tabela de venda), a função transforma esses dados para um formato adequado utilizando o “outputSellDto” (utilizado para fazer as conversões e deixar o dado do banco da forma necessária). Em seguida, ele identifica os produtos mais vendidos por cliente, agrupando as informações igual na lógica dos produtos por cliente. Após isso, ele ordena os produtos com base no número de pedidos, do maior para o menor, a função faz o cálculo do consumo médio e o valor médio gasto por cliente. Esses cálculos consideram o número de dias no mês e o valor médio gasto por cliente. Já para o gasto por cliente, ele faz uma soma do valor do produto a cada iteração no reduce.

Por fim, a função retorna um conjunto de objetos contendo informações sobre o consumo médio e o valor médio gasto por cliente nos produtos mais vendidos. Caso ocorra alguma exceção durante o processo, como falta de dados ou problemas internos, a função captura esse erro e lança uma exceção.

```

1  async function getAverageClientConsumptionUseCase(input) {
2    try {
3      const daysOfMonth = moment().daysInMonth();
4      const orders = await getSellByClientRepository(input.fk_client);
5
6      if (orders.length === 0) {
7        const error = new Error(
8          "Não é possível calcular o consumo médio desse cliente, pois ainda não foi vendido nenhum produto para esse cliente."
9        );
10       error.status = HttpStatus.BAD_REQUEST;
11       throw error;
12     }
13
14     const sellsDto = orders.map((item) => outputSellDto(item));
15
16     const bestSellingProducts = sellsDto.reduce((accumulator, currentValue) => {
17       const clientName = currentValue.client_name;
18       const productName = currentValue.product_name;
19
20       if (!accumulator[clientName]) {
21         accumulator[clientName] = [];
22       }
23
24       const existingProduct = accumulator[clientName].find(
25         (product) => product.product_name === productName
26       );
27
28       if (!existingProduct) {
29         accumulator[clientName].push({
30           orderRequests: 1,
31           client_name: clientName,
32           totalSale: currentValue.totalOrder,
33           product_name: productName,
34           product_value: currentValue.product_value,
35           orderProductQuantity: currentValue.orderProductQuantity,
36         });
37       } else {
38         existingProduct.orderRequests += 1;
39         existingProduct.totalSale += currentValue.totalOrder;
40         existingProduct.orderProductQuantity +=
41           currentValue.orderProductQuantity;
42       }
43
44       return accumulator;
45     }, {});
46
47     const bestSellingProductsArray = Object.values(bestSellingProducts).flat();
48
49     const orderBestSellingProducts = bestSellingProductsArray.sort(
50       (a, b) => b.orderRequests - a.orderRequests
51     );
52
53     const consumptionByClient = orderBestSellingProducts.reduce(
54       (acc, product) => {
55         const clientName = product.client_name;
56         if (!acc[clientName]) {
57           acc[clientName] = {
58             totalSales: 0,
59             totalOrders: 0,
60           };
61         }
62         acc[clientName].totalSales += product.totalSale;
63         acc[clientName].totalOrders += product.orderRequests;
64         return acc;
65       },
66       {}
67     );
68
69     const averageConsumption = Object.keys(consumptionByClient).map(
70       (client) => ({
71         client: client,
72         averageConsumption: (
73           consumptionByClient[client].totalOrders / daysOfMonth
74         ).toFixed(2),
75         averageSpent: (
76           consumptionByClient[client].totalSales /
77           consumptionByClient[client].totalOrders
78         ).toFixed(2),
79       })
80     );
81
82     return Object.values(averageConsumption);
83   } catch (error) {
84     const status = error.status || HttpStatus.INTERNAL_SERVER_ERROR;
85     throw {
86       message: `Erro ao listar os produtos por cliente: ${error.message}`,
87       status: status,
88     };
89   }
90 }

```