Campus Professor Barros

RELATÓRIO A3
Sistemas Distribuídos e Mobile

Salvador - BA
2023

Diego Pimenta dos Anjos
1272215402;
Fernando de Caires
Gonçalves 1272214754;
Lucca Cintra Poggio
1272219553;
Reinan Carvalho Amaral
12722126641;
Silvestre Carlos Elioterio Neto
12722116337;
Samuel Moura da Silva
12722129904.

DOCUMENTAÇÃO A3 SD

Trabalho da disciplina de Sistemas Distribuídos e Mobile. Neste documento iremos apresentar o nosso relatório referente à A3, na qual desenvolvemos uma aplicação Java utilizando Spring Boot para simular a gestão de vendas em uma loja de sapatos

Orientador: Adailton de Jesus Cerqueira Junior

Salvador - BA
2023

Introdução

Em uma loja de calçados, um aplicativo Java utilizando o framework Spring Boot será projetado para gerenciar vendas em um ambiente simulado. Nosso objetivo é aplicar os principais conceitos que envolvem gestão de clientes, controle de estoque e análise de vendas em um contexto prático e relevante.

A abordagem adotada foi construir um sistema completo, desde o cadastro do cliente até o reporte estatístico, utilizando o Spring Boot pela sua eficiência e facilidade de integração. O principal objetivo é fornecer funções essenciais, como cadastro, visualização, atualização e exclusão de clientes, gerenciamento de estoque com atividades de compras e geração de relatórios úteis para tomada de decisões.

O aplicativo será iniciado com os dados mínimos para permitir testes. Os objetivos incluem relatórios sobre os produtos mais populares, produtos por consumidor, consumo médio do cliente e itens faltantes no estoque. Esses documentos buscam fornecer dados valiosos para melhorar as operações comerciais. Fornecendo uma visão abrangente e detalhada do desempenho da loja, este aplicativo se torna uma ferramenta valiosa para proprietários e administradores de empresas.

Portanto, este projeto não só visa atender requisitos técnicos, mas também oferece soluções práticas para gestão de lojas de calçados, contribuindo para o sucesso da empresa através de uma visão clara e detalhada do desempenho da empresa.

Salvador - BA
2023

Fundamentação Teórica

A fundamentação teórica deste projeto inclui conceitos importantes que suportam decisões técnicas e estratégicas para desenvolvimento de aplicações de negócios em Java utilizando Spring Boot.

O sucesso de um projeto depende muito da seleção de sua linguagem de programação. Nesse caso, optamos pelo Java por ser amplamente aceito na indústria, robusto e portátil. Esta última qualidade, em particular, é significativa, pois permite que o aplicativo seja executado em diferentes ambientes sem a necessidade de recompilação, proporcionando assim, uma maior flexibilidade em integrações futuras.

Spring Boot foi escolhido como estrutura principal devido à sua popularidade, grande comunidade de desenvolvedores e capacidade de desenvolver aplicativos Java. Ele suporta a configuração e design de software, fornecendo um conjunto de bibliotecas para diversos problemas de desenvolvimento, como persistência de dados, segurança e integração com serviços externos. O processo de integração do Spring Boot acelera o desenvolvimento, permitindo que as equipes se concentrem em processos de negócios específicos.

A arquitetura de microsserviços foi introduzida para garantir elasticidade, escalabilidade e facilidade de manutenção. Cada função do aplicativo (gestão de clientes, estoque, vendas, relatórios contábeis) funciona como uma unidade separada, permitindo que diferentes equipes diferentes trabalhem juntas. Para aumentar a fiabilidade do sistema, é possível introduzir novos componentes de negócio; e isto, por sua vez, facilitará a estabilidade.

Manter a consistência e a integridade dos dados de vendas, estoque e clientes é fundamental no gerenciamento de vendas. Para garantir isso, um banco de dados relacional, especificamente MySQL, foi a melhor escolha. Os bancos de dados relacionais realizam o trabalho garantindo transações ACID, que oferecem Atomicidade, Consistência, Isolamento e Durabilidade. Esses critérios são necessários ao gerenciar dados confidenciais, como registros de vendas, estoque e informações de clientes.

Para melhorar a gestão e controle do sistema, integramos a tecnologia Docker em nosso desenvolvimento. Uma breve introdução ao docker; uma plataforma que facilita a construção, implantação e gerenciamento de aplicativos usando ferramentas. Um contêiner é um local remoto que contém tudo que um aplicativo precisa para ser executado, incluindo código, bibliotecas e dependências. Eles fornecem consistência e portabilidade, permitindo que aplicativos sejam executados em ambientes compatíveis com Docker.

Nossa equipe decidiu usar o Docker para construir a API em contêineres. Basicamente, fornecemos uma maneira simples e fácil para as pessoas testarem o que fazemos quando carregamos um arquivo em um repositório GitHub e incluímos um arquivo chamado docker-compose.yml. Ao baixar esses arquivos e executar o comando docker-compose up, todos os componentes necessários para usar a API são automaticamente atualizados e instalados porque tudo está configurado e configurado no contêiner de produção.

Este método oferece suporte ao processo de configuração de um ambiente de desenvolvimento, elimina possíveis diferenças entre configurações locais e permite que você comece a testar a API rapidamente. Esta é uma ótima maneira de garantir que outros desenvolvedores possam interagir e apoiar nossos projetos sem enfrentar obstáculos técnicos.

Dependências do Projeto

Maven: Uma ferramenta para gerenciamento de integrações e dependências, controle de integração, testes e distribuição de projetos Java.

Hibernate: estrutura Object Relational Mapper (ORM) que suporta armazenamento e recuperação de objetos em bancos de dados relacionais.

Spring boot Jpa, web, validation, test: Facilita a implementação de camadas de persistência, integrando-se ao Spring Data JPA para simplificar operações com o banco de dados. Oferece suporte ao desenvolvimento de aplicativos da web, proporcionando a base para a criação de endpoints REST. Provê suporte para a criação de testes unitários e de integração, garantindo a qualidade do código. Integra a validação de dados na aplicação, garantindo a integridade dos dados.

Mapstruct: Utilizados para simplificar o mapeamento entre objetos, permitindo uma manipulação mais eficiente dos dados.

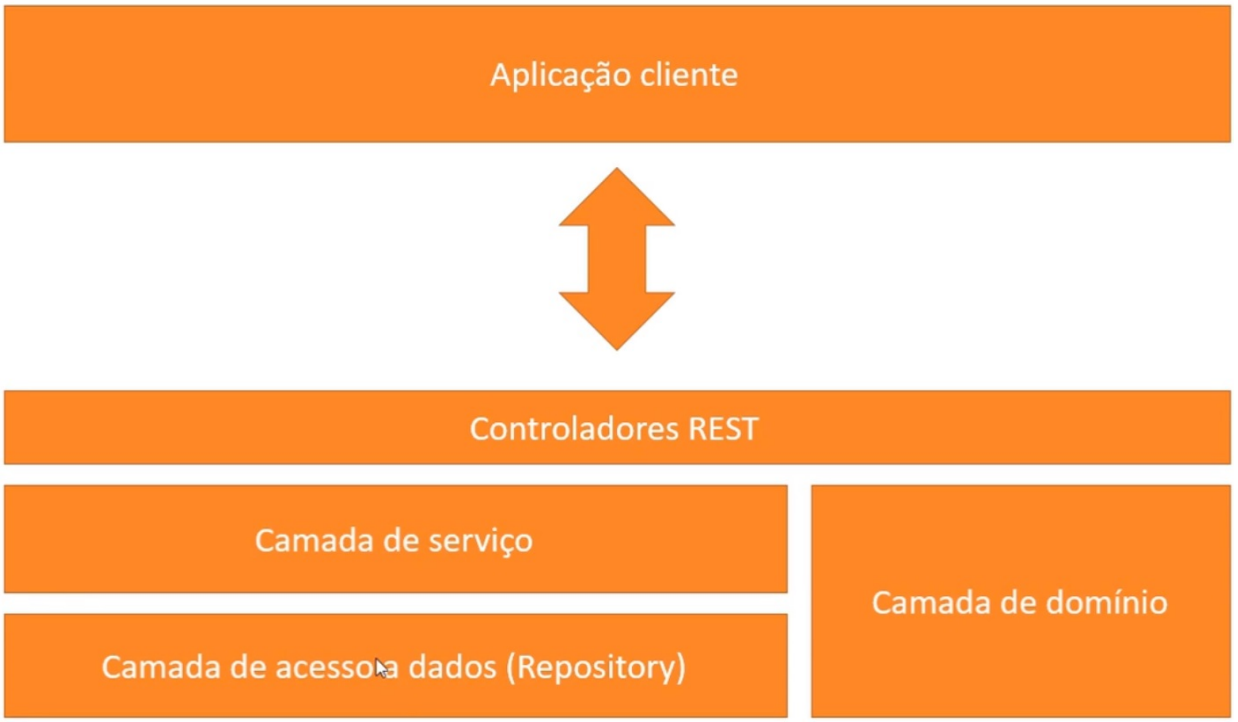
Mysql connector-j: Implementa o driver JDBC para comunicação com o banco de dados MySQL.

Lombok: Reduz a verbosidade do código Java, facilitando o desenvolvimento e a manutenção.

Todas as dependências listadas já estão em funcionamento dentro do docker, então ao executar o docker-compose.yml elimina a necessidade de baixar e configurar essas dependências para o funcionamento da api.

Projeto de Implementação

O sistema foi desenvolvido em camadas. Temos a camada Controladora REST representada no pacote **com.unifacs.nossopisanteapi.controller**, a camada de Domínio (Models) representada no pacote **com.unifacs.nossopisanteapi.model.entity/model.dto.request/model.dto.response** e as camadas de Serviço e Repositório (Repository ou DAO's) representadas nos pacotes **com.unifacs.nossopisanteapi.service/service.impl/service.mapper** e **com.unifacs.nossopisanteapi.repository** respectivamente.
A figura abaixo sumariza a arquitetura.



Salvador - BA
2023

Criar microsserviços é o objetivo principal da aplicação de gestão de vendas implementada em Java com Spring Boot. O sistema foi projetado para ser modular, escalável e eficiente, com cada componente individual servindo a um propósito único em relação à funcionalidade geral. A seguir, detalharemos cada um dos serviços principais.

1. Gerenciar Clientes

O Serviço de Clientes inclui funções CRUD (Create, Read, Update, Delete) para manipular dados do cliente. Com o Spring Data JPA, as tarefas contínuas podem ser simplificadas, permitindo que você se concentre no seu trabalho.

2. Gerenciar Estoque

Manter a precisão e integridade das informações do produto é responsabilidade do serviço de gerenciamento de estoque. As atualizações de estoque em tempo real ocorrem durante as vendas, evitando discrepâncias e garantindo dados atualizados. As operações CRUD também estão incluídas.

3. Gerenciar Vendas

O serviço de gerenciamento de vendas lida com o processamento dos pedidos de compra. Implementa as operações de CRUD para gerenciar as transações comerciais e, de forma crucial, comunica-se diretamente com o serviço de gerenciamento de estoque para garantir a consistência das informações. A utilização do Spring Boot facilita a criação de endpoints REST para interações externas com o serviço.

4. Geração de Relatórios Estatísticos

A geração de relatórios estatísticos é o foco principal deste serviço, pois fornece insights valiosos aos tomadores de decisão. Quando se trata de mapeamento de dados simples, bibliotecas como MapStruct tornam o processo muito mais fácil. O banco de dados é extraído de forma eficiente em busca de informações por meio do uso de consultas SQL avançadas.

Algoritmos

```java
package com.unifacs.nossopisanteapi.controller;

import com.unifacs.nossopisanteapi.model.dto.request.CreateProductStockRequestDto;

@RestController
@RequiredArgsConstructor
@RequestMapping("/product-stock")
public class ProductStockController {
    private final ProductStockService service;

    @PostMapping
    public ResponseEntity<ProductStockResponseDto> createProductStock(
        @RequestBody @Valid CreateProductStockRequestDto createProductStockRequestDto
    ) {
        return ResponseEntity.status(CREATED).body(service.createProductStock(createProductStockRequestDto));
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductStockResponseDto> getProductStock(@PathVariable("id") UUID stockId) {
        return ResponseEntity.ok(service.getProductStock(stockId));
    }

    @GetMapping
    public ResponseEntity<List<ProductStockResponseDto>> getAllProductsStock() {
        return ResponseEntity.ok(service.getAllProductsStock());
    }

    @PutMapping("/{id}")
    public ResponseEntity<ProductStockResponseDto> updateProductStock(
        @PathVariable("id") UUID stockId,
        @RequestBody @Valid UpdateProductStockRequestDto updateProductStockRequestDto
    ) {
        return ResponseEntity.ok(service.updateProductStock(stockId, updateProductStockRequestDto));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProductStock(@PathVariable("id") UUID stockId) {
        service.deleteProductStock(stockId);
        return ResponseEntity.noContent().build();
    }
}
```

Aqui temos os principais métodos de operação da classe ProductStockSevice:

createProductStock():

- Método para criar um novo produto de estoque.
- Recebe um objeto CreateProductStockRequestDto no corpo da requisição.
- Retorna uma resposta com o código de status CREATED e o corpo contendo as informações do novo produto.

getProductStock():

- Método para obter as informações de um produto de estoque específico.

- Recebe o ID do produto como parte da URL.
- Retorna uma resposta com as informações do produto ou um código de status adequado.

getAllProductStock():

- Método para obter informações de todos os produtos de estoque.
- Retorna uma resposta com uma lista de informações de produtos ou um código de status adequado.

updateProductStock():

- Método para atualizar as informações de um produto de estoque específico.
- Recebe o ID do produto como parte da URL e um objeto UpdateProductStockRequestDto no corpo da requisição.
- Retorna uma resposta com as informações atualizadas do produto ou um código de status adequado.

deleteProductStock():

- Método para excluir um produto de estoque específico.
- Recebe o ID do produto como parte da URL.
- Retorna uma resposta com um código de status adequado.

```java
package com.unifacs.nossopisanteapi.service.impl;

import com.unifacs.nossopisanteapi.model.dto.response.ReportConsumptionByClient;

@Service
@RequiredArgsConstructor
public class ReportServiceImpl implements ReportService {
    private final SaleRepository saleRepository;
    private final ProductStockRepository productStockRepository;
    private final ProductStockMapper  productStockMapper;
    private final ClientMapper clientMapper;

    @Override
    public ReportMostSoldProduct getMostSoldProduct() {
        var sales = saleRepository.findAll();

        var products = sales.stream()
            .map(Sale::getProducts)
            .flatMap(Collection::stream)
            .toList();

        var productsIds = products.stream()
            .map(ProductStock::getId)
            .toList();

        var map = productsIds.stream()
            .collect(Collectors.toMap(id -> Collections.frequency(productsIds, id), id -> id));

        var mostSoldProductId = map.get(Collections.max(map.keySet()));

        var mostSoldProduct = products.stream()
            .filter(product -> product.getId().equals(mostSoldProductId))
            .findFirst()
            .orElseThrow();

        return new ReportMostSoldProduct(productStockMapper.productStockToProductStockResponseDto(mostSoldProduct));
    }

    @Override
    public ReportProductByClient getProductByClient(UUID clientId) {
        var sales = saleRepository.findAll();

        var client = sales.stream()
            .filter(sale -> sale.getClient().getId().equals(clientId))
```

```
65              .map(clientMapper::clientToClientResponseDto)
66              .orElseThrow();
67
68          var productsByClient = sales.stream()
69              .filter(sale -> sale.getClient().getId().equals(clientId))
70              .map(Sale::getProducts)
71              .flatMap(Collection::stream)
72              .map(productStockMapper::productStockToProductStockResponseDto)
73              .toList();
74
75          return new ReportProductByClient(client, productsByClient);
76      }
77
78      @Override
79      public ReportConsumptionByClient getConsumptionByClient(UUID clientId) {
80          var sales = saleRepository.findAll();
81
82          var client = sales.stream()
83          .filter(sale -> sale.getClient().getId().equals(clientId))
84          .findFirst()
85          .map(Sale::getClient)
86          .map(clientMapper::clientToClientResponseDto)
87          .orElseThrow();
88
89          var averageConsumptionByClient = sales.stream()
90              .filter(sale -> sale.getClient().getId().equals(clientId))
91              .map(Sale::getProducts)
92              .flatMap(Collection::stream)
93              .map(ProductStock::getPrice)
94              .reduce(BigDecimal.ZERO, BigDecimal::add);
95
96          return new ReportConsumptionByClient(client, averageConsumptionByClient);
97      }
98
99      @Override
100     public ReportLowStockProducts getLowStockProducts() {
101         var productsStock = productStockRepository.findAll();
102
103         var lowStockProducts = productsStock.stream()
104             .filter(productStock -> productStock.getQuantity() < 10)
105             .map(productStockMapper::productStockToProductStockResponseDto)
106             .toList();
107
108         return new ReportLowStockProducts(lowStockProducts);
```

Essa é uma classe de implementação de um serviço de relatórios na nossa aplicação:

getMostSoldProduct():

- Utiliza o método findAll() do saleRepository para obter todas as vendas.
- Usa a API de Stream para transformar a lista de vendas em uma lista de produtos vendidos (products).
- Obtém uma lista de IDs de produtos (productsIds).
- Cria um mapa onde a chave é a frequência de cada produto e o valor é o ID do produto.
- Identifica o ID do produto mais vendido com base na maior frequência.

- Recupera o produto mais vendido com base no ID.
- Converte o produto para um DTO de resposta e retorna um relatório.

getProductByClient():

- Similar a getMostSoldProduct, utiliza findAll() para obter todas as vendas.
- Filtra as vendas para encontrar aquelas associadas ao cliente específico (clientId).
- Obtém informações do cliente associado à primeira venda encontrada.
- Utiliza a API de Stream para obter uma lista de produtos vendidos a esse cliente.
- Converte os produtos para DTOs de resposta.
- Retorna um relatório contendo as informações do cliente e os produtos comprados por ele.

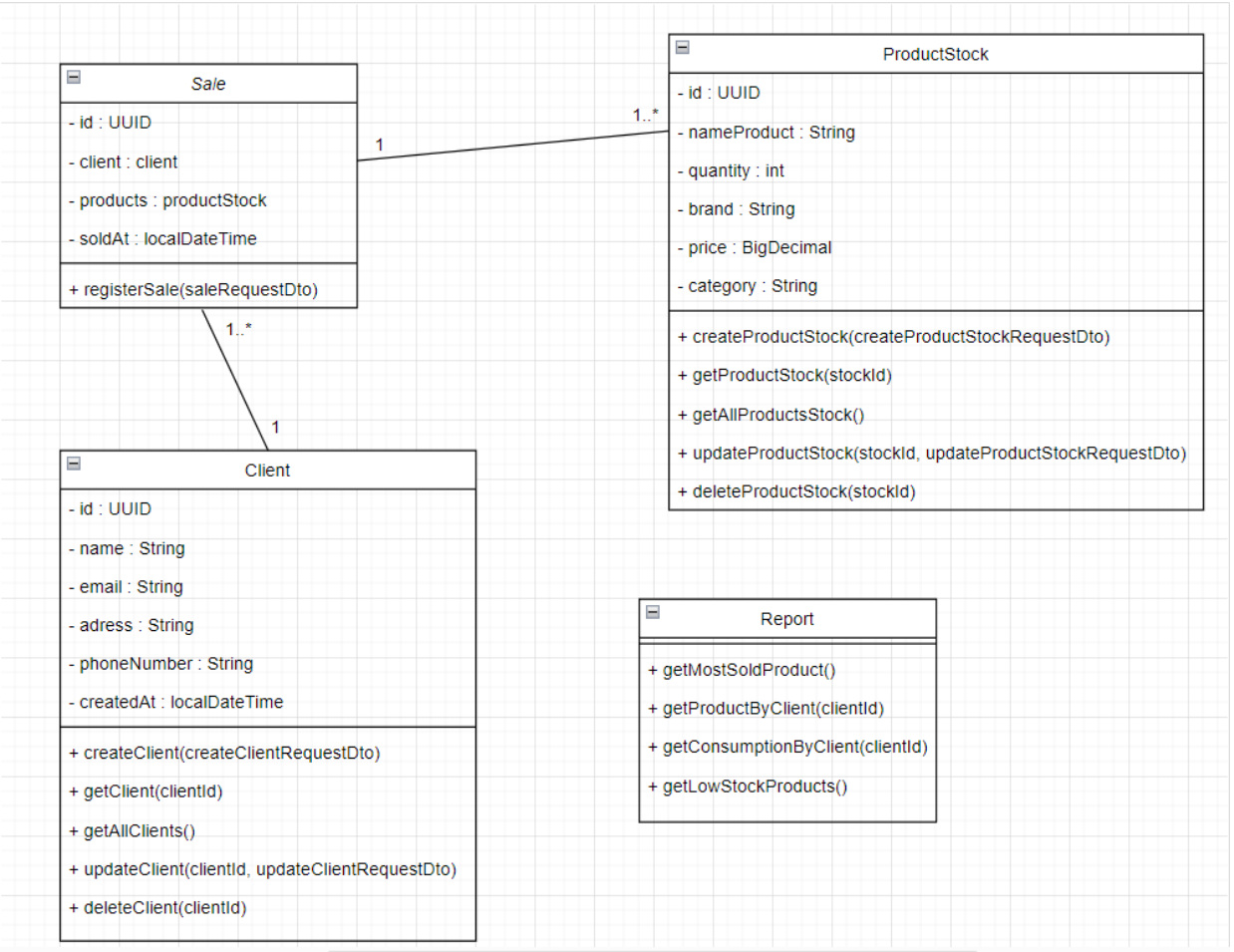
getConsumptionByClient():

- Novamente, utiliza findAll() para obter todas as vendas.
- Filtra as vendas para encontrar aquelas associadas ao cliente específico (clientId).
- Obtém informações do cliente associado à primeira venda encontrada.
- Utiliza a API de Stream para obter uma lista de preços dos produtos vendidos a esse cliente.
- Usa a operação reduce para somar os preços, obtendo o total de consumo.
- Retorna um relatório contendo as informações do cliente e o consumo médio.

getLowStockProducts():

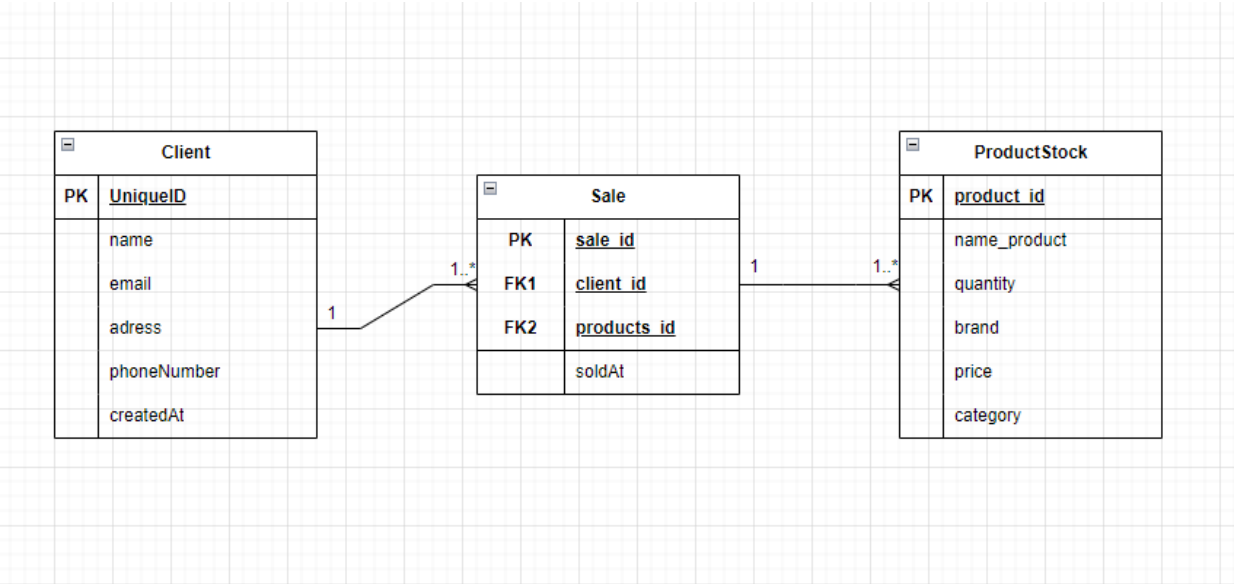
- Utiliza findAll() para obter todos os produtos no estoque.
- Filtra os produtos para encontrar aqueles com uma quantidade em estoque inferior a 10.
- Converte os produtos de baixo estoque para DTOs de resposta.
- Retorna um relatório contendo os produtos em baixo estoque.

Diagrama de Classe

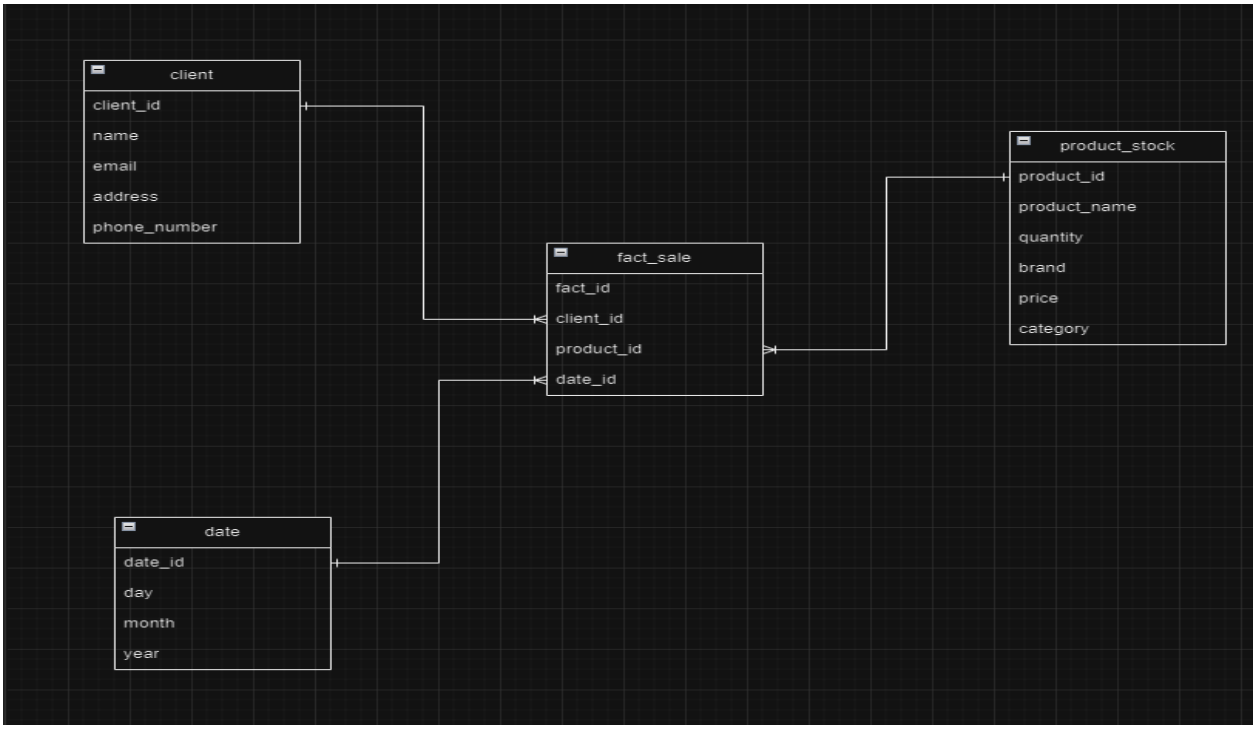


Salvador - BA
2023

Diagrama Entidade-Relacionamento



Modelagem Dimensional



Salvador - BA
2023

Considerações Finais

No âmbito deste projeto, desenvolvemos uma aplicação prática utilizando o framework Spring Boot em Java, com o objetivo de simplificar a gestão de vendas de uma loja de calçados. Desenvolvido com. A escolha deste contexto foi motivada pela sua relevância e pela possibilidade de aplicação de conceitos básicos de gestão de clientes, gestão de estoques e análise de vendas.

Construímos um sistema completo desde o cadastro de clientes até relatórios estatísticos, com foco em recursos essenciais como CRUD do cliente, gerenciamento de estoque e relatórios informativos.

Nosso objetivo não é apenas atender às suas necessidades técnicas. Queremos oferecer soluções práticas para administrar uma sapataria. A nossa aplicação pretende contribuir para o sucesso das empresas, proporcionando uma visão clara e abrangente do seu desempenho.

Enfrentamos desafios como configurar o MySQL no Docker e escolhemos o Spring Boot por sua eficiência e facilidade de integração. Adotamos a arquitetura de microsserviços para modularidade e escalabilidade, utilizamos ferramentas como Mapstruct para mapeamento de objetos, Spring Boot JPA para persistência de dados, MySQL Connector-J para comunicação com bancos de dados MySQL e também o Lombok reduzir a redundância de código e facilitar o desenvolvimento.

Utilizamos os serviços de gestão de clientes, gestão de stocks, processamento de vendas e relatórios estatísticos, os quais desempenha funções fundamentais na aplicação

Estamos planejando melhorias futuras, incluindo melhorias de segurança e relatórios e testes aprimorados para garantir um desempenho consistente.

Salvador - BA
2023

Bibliografia

Oracle. (n.d.). Java Programming Language. Recuperado em 15 de dezembro de 2023, de https://www.oracle.com/java/

Hibernate. (n.d.). Hibernate ORM. Acessado em 15 de dezembro de 2023, disponível em https://hibernate.org/orm/

MapStruct. (n.d.). MapStruct - Simplifying Object Mapping in Java. Recuperado em 15 de dezembro de 2023, de https://mapstruct.org/

MySQL. (n.d.). MySQL Database. Acessado em 15 de dezembro de 2023, disponível em https://www.mysql.com/

Docker Documentation. (n.d.). Get Started with Docker. Recuperado em 15 de dezembro de 2023, de https://docs.docker.com/get-started/

Lombok Project. (n.d.). Project Lombok. Acessado em 15 de dezembro de 2023, disponível em https://projectlombok.org/

Apache Maven. (n.d.). Apache Maven. Recuperado em 15 de dezembro de 2023, de https://maven.apache.org/

Spring Data JPA. (n.d.). Spring Data JPA - Simplifies Database Access in Java. Acessado em 15 de dezembro de 2023, disponível em https://spring.io/projects/spring-data-jpa

Spring Boot. (n.d.). Spring Boot - Simplify Java Development. Recuperado em 15 de dezembro de 2023, de https://spring.io/projects/spring-boot

Baeldung. (n.d.). REST with Spring Boot. Recuperado em 15 de dezembro de 2023, de https://www.baeldung.com/rest-with-spring-series