

Lucas Juciel Portes de Oliveira

Proposta de Microservice:

## Serviço de Recomendação Personalizada

### Definição dos Bounded Contexts e Aggregates

#### **Bounded Context:** Recomendação de Serviços

O bounded context para o domínio de Recomendação de Serviços gerencia a lógica e os dados relacionados à geração e entrega de recomendações personalizadas de serviços aos usuários com base em seu comportamento, histórico e preferências. Suas funcionalidades principais se estendem em:

#### **Análise de Dados de Usuário:**

Coleta e processamento de dados de usuários, incluindo histórico de interações, preferências declaradas, informações demográficas e atividades recentes.

#### **Modelagem de Perfis de Usuário:**

Desenvolvimento de perfis de usuário com base nos dados coletados, identificando padrões de comportamento, interesses e necessidades individuais.

#### **Geração de Recomendações:**

Utilização de algoritmos de recomendação para gerar sugestões personalizadas de serviços que possam ser do interesse do usuário. Consideração de diversos fatores, como histórico de compras, navegação recente, tendências de mercado e preferências semelhantes de outros usuários.

#### **Apresentação de Recomendações:**

Entrega de recomendações aos usuários por meio de diferentes canais, como recomendações na página inicial, e-mails personalizados, notificações push e a personalização das recomendações com base no contexto do usuário, dispositivo e horário de acesso.

#### **Feedback do Usuário:**

Coleta de feedback dos usuários sobre as recomendações recebidas para refinamento contínuo do modelo de recomendação. Análise de métricas de engajamento para avaliar a eficácia das recomendações e ajustar os algoritmos conforme necessário.

### Aggregates

#### **Aggregate:** Recomendação

**Identificador de Recomendação (ID):** A raiz do aggregate, único para cada recomendação.

Informações do Serviço Recomendado: Incluindo nome, descrição, categoria etc.

**Detalhes da Recomendação:** Critérios específicos usados para gerar a recomendação, como histórico de interações, preferências do usuário, tendências de mercado etc.

**Feedback do Usuário:** Dados relacionados ao feedback fornecido pelo usuário em relação à recomendação recebida.

**Data e Hora da Recomendação:** Para rastrear quando a recomendação foi gerada.

Este aggregate de Recomendação encapsula todas as informações relacionadas a uma recomendação específica feita para um usuário. A raiz do aggregate é o identificador único da recomendação, e todas as operações relacionadas à recomendação, como geração, entrega e feedback, são realizadas através dessa raiz.

#### **Benefícios:**

**Consistência dos Dados:** Garante que todas as informações relacionadas a uma recomendação estejam agrupadas e consistentes.

**Aplicação de Regras de Negócio:** Permite a implementação eficaz de lógica de negócio relacionada à geração, entrega e análise de recomendações.

**Manutenção da Integridade do Modelo de Domínio:** Facilita a persistência coerente dos dados e a aplicação de regras de negócio complexas.

#### **Tecnologias Utilizadas:**

**Framework de Agregação:** Pode-se utilizar frameworks de desenvolvimento que suportem a definição e implementação de aggregates, como o Spring Framework em Java ou o Django em Python.

**Bancos de Dados:** Para armazenamento persistente dos aggregates, podem ser utilizados bancos de dados relacionais ou NoSQL, dependendo dos requisitos de escalabilidade e consistência dos dados.

## Implementação do Aggregate de Recomendação

Utilizando a assistência do Chat GPT, foi desenvolvido a implementação do aggregate de Recomendação no contexto de Recomendação de Serviços, utilizando Java e seguindo as práticas da Clean Architecture:

```

1
2 // Entidade: Recomendação
3 public class Recomendacao {
4     private Long id;
5     private ServicoRecomendado servico;
6     private DetalhesRecomendacao detalhes;
7     private LocalDateTime dataHora;
8
9     // Getters e Setters
10 }
11
12 // Objeto de Valor: Detalhes da Recomendação
13 public class DetalhesRecomendacao {
14     private Critérios criterios;
15     private Feedback feedback;
16
17     // Construtor, Getters e Setters
18 }
19
20 // Objeto de Valor: Critérios
21 public class Critérios {
22     private String historicoInteracoes;
23     private String preferenciasUsuario;
24     private String tendenciasMercado;
25
26     // Construtor, Getters e Setters
27 }
28
29 // Objeto de Valor: Feedback
30 public class Feedback {
31     private String comentario;
32     private int avaliacao;
33
34     // Construtor, Getters e Setters
35 }
36
37 // Repositório: RecomendacaoRepository (interface)
38 public interface RecomendacaoRepository {
39     Recomendacao findById(Long id);
40     void save(Recomendacao recomendacao);
41     void delete(Recomendacao recomendacao);
42 }
43
44 // Implementação do Repositório: RecomendacaoRepositoryImpl
45 public class RecomendacaoRepositoryImpl implements RecomendacaoRepository {
46     // Implementação dos métodos de acesso a dados
47 }
48

```

Imagem 1 – Implementação do DDD.

## Modelagem dos Domínios

Entidade do Produto e Objetos de Valor

Atribuindo a entidade produtos e objetos de valor, podemos identificar as seguintes situações.

```
// Entidade Recomendação
import java.time.LocalDateTime;

public class Recomendacao {
    private Long id;
    private ServicoRecomendado servicoRecomendado; // ServicoRecomendado é uma referência ao serviço que está sendo recomendado.
    private DetalhesRecomendacao detalhes; // DetalhesRecomendacao encapsula os detalhes específicos da recomendação.
    private LocalDateTime dataHora; /// LocalDateTime dataHora armazena a data e hora em que a recomendação foi feita.

    // Construtor, Getters e Setters
}

// Objeto de Valor: Detalhes da Recomendação
public class DetalhesRecomendacao {
    private Critérios criterios;
    private Feedback feedback;

    // Construtor, Getters e Setters
}

// Objeto de Valor: Critérios
public class Critérios {
    private String historicoInteracoes;
    private String preferenciasUsuario;
    private String tendenciasMercado;

    // Construtor, Getters e Setters
}

// Objeto de Valor: Feedback
public class Feedback {
    private String comentario;
    private int avaliacao;

    // Construtor, Getters e Setters
}
```

Imagem 2 – Desenvolvimento de Entidade Produto.

## Regras de Negócio para Recomendação de Serviços

O sucesso de um sistema de recomendação de serviços depende diretamente da implementação de regras de negócio essenciais, oferecendo recomendações relevantes e personalizadas que atendam às suas necessidades e interesses.

### Geração de Recomendações:

As recomendações devem ser geradas com base em critérios específicos, como histórico de interações do usuário, preferências declaradas e tendências de mercado. Também devem ser relevantes e personalizadas para cada usuário, levando em consideração suas características individuais e comportamento passado.

### Apresentação de Recomendações:

As recomendações devem ser apresentadas ao usuário de maneira clara e atraente, através de diferentes canais, como página inicial, e-mails personalizados ou notificações push. A

apresentação das recomendações deve ser adaptada ao contexto do usuário, considerando o dispositivo utilizado, horário de acesso e histórico de interações.

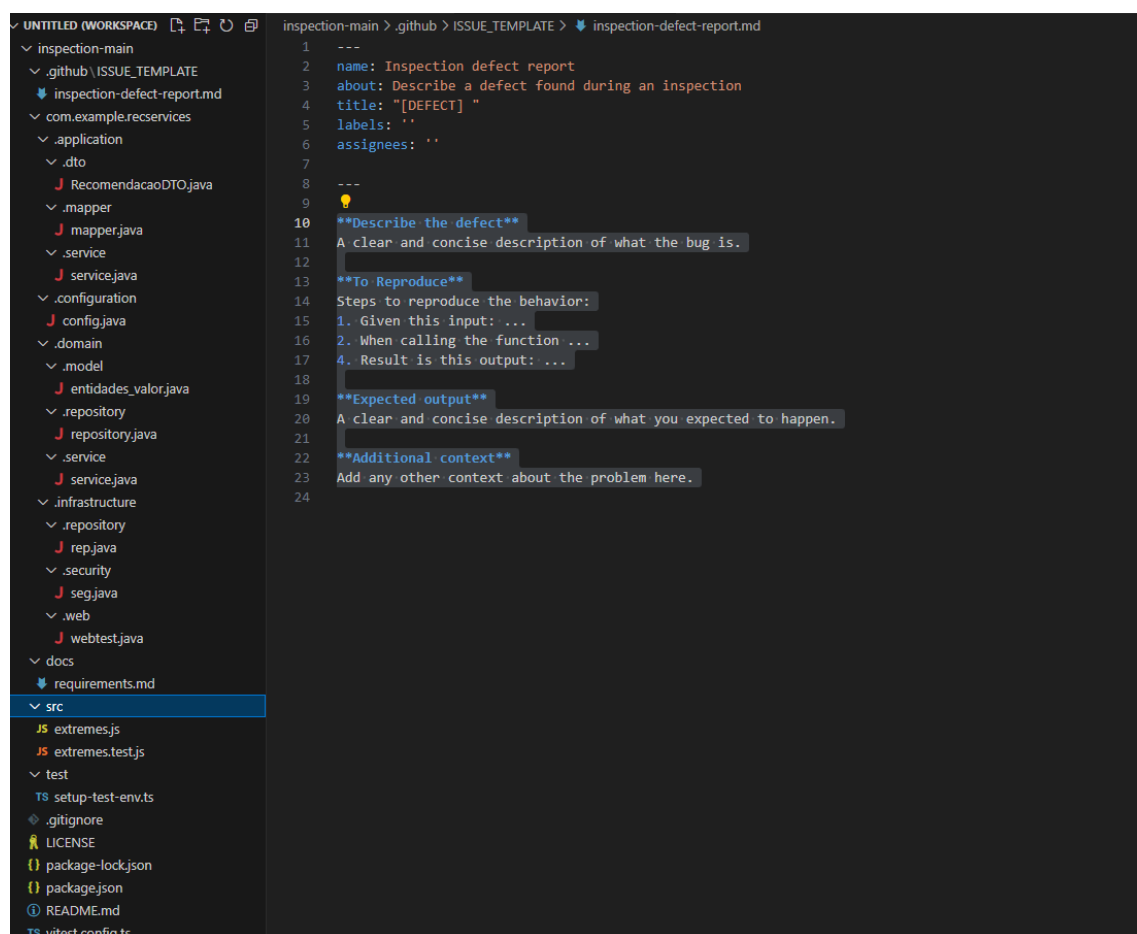
### Feedback do Usuário:

Deve ser coletado feedback dos usuários sobre as recomendações recebidas, incluindo comentários e avaliações. O feedback dos usuários deve ser analisado para avaliar a eficácia das recomendações e identificar áreas para melhorias no sistema de recomendação.

### Atualização Contínua do Modelo de Recomendação:

O modelo de recomendação deve ser constantemente atualizado e refinado com base no feedback dos usuários e nas mudanças no comportamento e preferências deles. Novos algoritmos e técnicas de recomendação devem ser explorados e implementados para melhorar a precisão e relevância das recomendações.

## Estrutura de Pacotes



The image shows a code editor with a project structure on the left and a template for an inspection defect report on the right.

**Project Structure (Left Panel):**

- inspection-main
  - .github\ISSUE\_TEMPLATE
    - inspection-defect-report.md
  - com.example.recservices
    - .application
      - .dto
        - RecomendacaoDTO.java
      - .mapper
        - mapper.java
      - .service
        - service.java
      - .configuration
        - config.java
      - .domain
        - .model
          - entidades\_valor.java
      - .repository
        - repository.java
      - .service
        - service.java
      - .infrastructure
        - .repository
          - rep.java
        - .security
          - seg.java
        - .web
          - webtest.java
      - docs
        - requirements.md
      - src
        - extremes.js
        - extremes.test.js
        - test
          - setup-test-env.ts
        - .gitignore
        - LICENSE
        - package-lock.json
        - package.json
        - README.md
        - vitest.config.ts

**Template Content (Right Panel):**

```
1 ---
2 name: Inspection defect report
3 about: Describe a defect found during an inspection
4 title: "[DETECT] "
5 labels: ''
6 assignees: ''
7 ---
8
9
10 **Describe the defect**
11 A clear and concise description of what the bug is.
12
13 **To Reproduce**
14 Steps to reproduce the behavior:
15 1. Given this input: ...
16 2. When calling the function ...
17 4. Result is this output: ...
18
19 **Expected output**
20 A clear and concise description of what you expected to happen.
21
22 **Additional context**
23 Add any other context about the problem here.
24
```

Imagem 3 – Estrutura de Pacotes.

## Desmistificando a arquitetura de referência da Clean Architecture x Estrutura de pacotes do Java

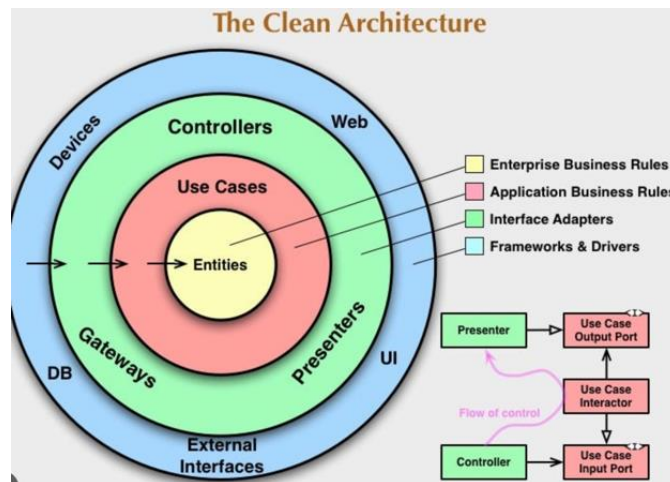


Imagem 4 – Clean Architecture.

Ao atribuir no processo de recomendação sobre o Clean Architecture, é identificado a presença de 5 temas necessários, são eles:

- Entities
- Use Cases
- Interface Adapters
- Frameworks & Drivers
- Devices, Web, UI, DB, External Interfaces
- Organização dos Pacotes

O objetivo principal desses princípios é a separação de interesses, tornando os sistemas mais fáceis de gerenciar, adaptar e manter. Com o baixo nível de proficiência nas ferramentas utilizadas, com o auxílio da ferramenta GPT, foram desenvolvidos os seguintes comandos.

Camada de Domínio (domain)

Passo 1: Criar a Entidade Produto

```
package com.example.recommendations.domain.model;

public class Recomendacao {
    private Long id;
```

```
private String servicoRecomendado;
private String criterios;
private String feedback;

// Construtor, Getters e Setters

public Recomendacao(Long id, String servicoRecomendado, String
criterios, String feedback) {
    this.id = id;
    this.servicoRecomendado = servicoRecomendado;
    this.criterios = criterios;
    this.feedback = feedback;
}

// Getters e Setters

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getServicoRecomendado() {
    return servicoRecomendado;
}

public void setServicoRecomendado(String servicoRecomendado) {
    this.servicoRecomendado = servicoRecomendado;
}

public String getCriterios() {
    return criterios;
}

public void setCriterios(String criterios) {
    this.criterios = criterios;
}

public String getFeedback() {
    return feedback;
}

public void setFeedback(String feedback) {
    // Verifica se o feedback está vazio e lança uma exceção
    if (feedback == null || feedback.trim().isEmpty()) {
        throw new IllegalArgumentException("Feedback não pode ser
vazio");
    }
}
```

```

    }
    this.feedback = feedback;
}
}

```

## Passo 2 Criar o Repository

```

package com.example.recommendations.domain.repository;

import com.example.recommendations.domain.model.Recomendacao;

import java.util.List;
import java.util.Optional;

public interface RecomendacaoRepository {
    Optional<Recomendacao> findById(Long id);
    List<Recomendacao> findAll();
    Recomendacao save(Recomendacao recomendacao);
    void deleteById(Long id);
}

```

## Camada de Aplicação (application)

### Passo 1: Definir os DTOs (Data Transfer Objects)

```

////

package com.example.recommendations.application.dto;

public class RecomendacaoDTO {
    private Long id;
    private String servicoRecomendado;
    private String criterios;
    private String feedback;

    public RecomendacaoDTO(Long id, String servicoRecomendado, String
criterios, String feedback) {
        this.id = id;
        this.servicoRecomendado = servicoRecomendado;
        this.criterios = criterios;
        this.feedback = feedback;
    }

    public Long getId() {
        return id;
    }
}

```



```

    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getServicoRecomendado() {
        return servicoRecomendado;
    }

    public void setServicoRecomendado(String servicoRecomendado) {
        this.servicoRecomendado = servicoRecomendado;
    }

    public String getCritérios() {
        return criterios;
    }

    public void setCritérios(String criterios) {
        this.criterios = criterios;
    }

    public String getFeedback() {
        return feedback;
    }

    public void setFeedback(String feedback) {
        // Verifica se o feedback está vazio e lança uma exceção
        if (feedback == null || feedback.trim().isEmpty()) {
            throw new IllegalArgumentException("Feedback não pode ser
vazio");
        }
        this.feedback = feedback;
    }
}

```

## Passo 2: Criar Mappers

```

//

// Em seu ProdutoService.java
@Autowired
public ProdutoService(ProdutoRepository produtoRepository, ProdutoMapper
produtoMapper) {
    this.produtoRepository = produtoRepository;
    this.produtoMapper = produtoMapper;
}

```

```

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class ProdutoServiceTest {

    @Autowired
    private ProdutoService produtoService;

    // Aqui você deve implementar os testes para os métodos do serviço
    // Exemplo de teste:
    @Test
    public void testListarProdutos() {
        // Implemente o teste aqui
    }
}

```

### Passo 3: Implementar Casos de Uso

```

////

package com.example.recommendations.application.service;

import com.example.recommendations.domain.model.Produto;
import com.example.recommendations.domain.repository.ProdutoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ProdutoService {

    private final ProdutoRepository produtoRepository;

    @Autowired
    public ProdutoService(ProdutoRepository produtoRepository) {
        this.produtoRepository = produtoRepository;
    }

    public List<Produto> listarProdutos() {

```

```

        return produtoRepository.findAll();
    }

    public Produto buscarProdutoPorId(Long id) {
        return produtoRepository.findById(id)
            .orElseThrow(() -> new IllegalArgumentException("Produto
não encontrado"));
    }

    public Produto salvarProduto(Produto produto) {
        // Aqui você pode adicionar lógica de validação ou processamento
        adicional antes de salvar o produto
        return produtoRepository.save(produto);
    }

    public void deletarProduto(Long id) {
        produtoRepository.deleteById(id);
    }
}

```

#### Passo 4: Configurar Injeção de Dependências

```

package com.example.recommendations.application.service;

import com.example.recommendations.application.dto.RecomendacaoDTO;
import com.example.recommendations.application.mapper.RecomendacaoMapper;
import com.example.recommendations.domain.model.Recomendacao;
import com.example.recommendations.domain.service.RecomendacaoService;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class RecomendacaoAppService {

    private final RecomendacaoService recomendacaoService;
    private final RecomendacaoMapper recomendacaoMapper;

    public RecomendacaoAppService(RecomendacaoService
recomendacaoService, RecomendacaoMapper recomendacaoMapper) {
        this.recomendacaoService = recomendacaoService;
        this.recomendacaoMapper = recomendacaoMapper;
    }

    public List<RecomendacaoDTO> listarTodasRecomendacoes() {
        List<Recomendacao> recomendacoes =
recomendacaoService.listarTodasRecomendacoes();
        return recomendacoes.stream()

```

```

        .map(recomendacaoMapper::toDTO)
        .collect(Collectors.toList());
    }

    public RecomendacaoDTO buscarRecomendacaoPorId(Long id) {
        Recomendacao recomendacao =
recomendacaoService.buscarRecomendacaoPorId(id);
        return recomendacaoMapper.toDTO(recomendacao);
    }

    public RecomendacaoDTO salvarRecomendacao(RecomendacaoDTO
recomendacaoDTO) {
        Recomendacao recomendacao =
recomendacaoMapper.toEntity(recomendacaoDTO);
        Recomendacao savedRecomendacao =
recomendacaoService.salvarRecomendacao(recomendacao);
        return recomendacaoMapper.toDTO(savedRecomendacao);
    }

    public void deletarRecomendacao(Long id) {
        recomendacaoService.deletarRecomendacao(id);
    }
}

```

Camada de Infraestrutura (infrastructure)

Passo 1: Configurar Repositórios de Dados

```

///

package com.example.recommendations.infrastructure.repository;

import com.example.recommendations.domain.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    // Você pode adicionar métodos de consulta personalizados aqui, se
necessário
}

```

Passo 2: Implementar Controladores REST

```

///

```

```

package com.example.recommendations.infrastructure.web;

import com.example.recommendations.application.dto.ProdutoDTO;
import com.example.recommendations.application.service.ProdutoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/produtos")
public class ProdutoController {

    private final ProdutoService produtoService;

    @Autowired
    public ProdutoController(ProdutoService produtoService) {
        this.produtoService = produtoService;
    }

    @GetMapping
    public ResponseEntity<List<ProdutoDTO>> listarProdutos() {
        List<ProdutoDTO> produtos = produtoService.listarProdutos();
        return ResponseEntity.ok(produtos);
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProdutoDTO> buscarProdutoPorId(@PathVariable
Long id) {
        ProdutoDTO produto = produtoService.buscarProdutoPorId(id);
        return ResponseEntity.ok(produto);
    }

    @PostMapping
    public ResponseEntity<ProdutoDTO> salvarProduto(@RequestBody
ProdutoDTO produtoDTO) {
        ProdutoDTO produtoSalvo =
produtoService.salvarProduto(produtoDTO);
        return
ResponseEntity.status(HttpStatus.CREATED).body(produtoSalvo);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deletarProduto(@PathVariable Long id) {
        produtoService.deletarProduto(id);
        return ResponseEntity.noContent().build();
    }
}

```

```
}  
}
```

## Configurar Banco de Dados e Transações

### Passo 1: Configurar o Banco de Dados em Memória H2

```
<!-- Maven -->  
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

```
# application.properties  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.h2.console.enabled=true  
  
# application.yml  
spring:  
  datasource:  
    url: jdbc:h2:mem:testdb  
    driverClassName: org.h2.Driver  
    username: sa  
    password: password  
  h2:  
    console:  
      enabled: true
```