Nome do projeto: Beauty Book

Ângelo Victor de Lima

O Problema

O projeto visa resolver a dificuldade de gerenciamento eficiente de agendamentos e relacionamento com clientes em estabelecimentos de beleza e bem-estar. Muitos profissionais ainda utilizam métodos manuais, o que leva a erros, falhas de comunicação, perda de agendamentos e baixa fidelização. O sistema BeautyBook surge como uma solução tecnológica completa para atender a esses desafios, proporcionando uma experiência melhor tanto para os clientes quanto para os profissionais.

Funcionalidades Principais

Cadastro e Gerenciamento de Estabelecimentos
Os estabelecimentos podem se cadastrar informando nome, endereço, localização geográfica, serviços oferecidos, foto, profissionais vinculados e horários de funcionamento. Cada cadastro é persistido com validações e vinculações reais entre as entidades.

Cadastro de Profissionais

Cada profissional pode ser cadastrado com nome, telefone, email, cpf e valor de serviço padrão. A vinculação com os estabelecimentos é feita pelo cadastro ou edição de estabelecimentos.

Agendamento de Serviços
O sistema permite que clientes agendem atendimentos com profissionais específicos em estabelecimentos previamente cadastrados, informando:

- Data da reserva (LocalDate)
- Horário de início (LocalTime)
- Identificador do serviço, profissional e estabelecimento

Antes da confirmação do agendamento, o sistema executa **validações rigorosas** para garantir consistência e viabilidade do agendamento:

1. **Validação de Entidades Existentes**
   O sistema verifica se todos os IDs fornecidos (cliente, profissional, estabelecimento e serviço) são válidos e existentes, acessando diretamente os respectivos microserviços por meio dos gateways de integração.
2. **Verificação de Funcionamento do Estabelecimento**
    - Através do EstablishmentGateway, o sistema obtém os horários de funcionamento para o dia da semana correspondente.

    - Garante que o horário inicial do agendamento esteja dentro do intervalo de funcionamento do estabelecimento.

3. **Verificação de Disponibilidade do Profissional**
    - O sistema acessa os dados de disponibilidade do profissional no dia da semana informado.
    - Valida se o horário do agendamento está dentro da janela de disponibilidade registrada.
4. **Verificação de Conflitos de Horário**
    - Com base na duração do serviço (também consultada dinamicamente via EstablishmentGateway), o sistema calcula a hora final do atendimento.
    - Compara o intervalo do novo agendamento com os agendamentos existentes do profissional para o mesmo dia.
    - Se houver sobreposição de horários, lança uma exceção BookingConflictException.
5. **Notificação do Agendamento**
   Após passar por todas as validações, o agendamento é salvo com status SCHEDULED e uma notificação é disparada via BookingNotifierGateway.

Cancelamento e Reagendamento de Agendamentos
Os clientes podem cancelar ou reagendar seus agendamentos. Assim como a criação do agendamento, possuí diversas regras de negócio para garantir a integridade das agendas.

Sistema de Avaliações com Controle de Unicidade
Após a conclusão do serviço, o cliente pode avaliar o estabelecimento com uma nota de 1 a 5 estrelas e um comentário. O sistema garante que cada cliente só possa avaliar um estabelecimento uma única vez.

Busca e Filtragem de Estabelecimentos
O sistema oferece uma busca avançada de estabelecimentos com os seguintes filtros:

Nome do estabelecimento

Nome do serviço

Localização

Nota mínima de avaliação

Faixa de preço (mínima e máxima)

Dia da semana de funcionamento

Listagem e Gerenciamento de Agendamentos
Estabelecimentos podem listar todos os agendamentos, filtrando por período, status e data. Essa funcionalidade permite a gestão precisa da agenda, incluindo controle de reagendamentos e cancelamentos.

Geração de Arquivos .ics para Sincronização com Calendários
O sistema permite o download de arquivos .ics (formato padrão de calendário) para

que clientes e profissionais possam importar seus agendamentos diretamente no Google Calendar, Outlook ou Apple Calendar.

Endpoints e Arquitetura de Acesso

O sistema é composto por quatro microserviços independentes, cada um com seu próprio Swagger UI para documentação e testes. Além disso, utiliza o Eureka Server para descoberta de serviços e o API Gateway para unificação das rotas de acesso.

Documentação Swagger

Cada microserviço expõe sua própria interface Swagger gerada automaticamente com SpringDoc:

Microserviço	Porta	URL do Swagger
Customer Management	**8081**	**http://localhost:8081/swagger-ui/index.html**
Establishment Management	**8082**	**http://localhost:8082/swagger-ui/index.html**
Professional Management	**8083**	**http://localhost:8083/swagger-ui/index.html**
Booking API	**8084**	**http://localhost:8084/swagger-ui/index.html**

Cada um desses endpoints permite a visualização dos contratos da API, com exemplos, schemas, e possibilidade de testar diretamente os endpoints REST.

API Gateway (porta 8080)

O API Gateway centraliza o acesso a todos os microserviços, servindo como uma única entrada para a aplicação. Ele não possui Swagger próprio, pois sua responsabilidade é apenas de roteamento.

Com o uso do Spring Cloud Gateway + Eureka Discovery, as rotas são registradas dinamicamente com base no spring.application.name de cada microserviço.

Exemplos de acesso via gateway:

http://localhost:8080/customer-management/customers

http://localhost:8080/establishment-management/establishments

http://localhost:8080/professional-management/professionals

http://localhost:8080/booking/bookings

O gateway busca o nome de cada serviço no Eureka Server e roteia automaticamente a requisição para a instância correspondente.

Eureka Server (porta 8761)

O Eureka Server atua como registro de serviços. Ao iniciar, cada microserviço se registra automaticamente, permitindo:

- Descoberta automática de serviços pelo gateway e por outros microserviços.
- Alta flexibilidade e resiliência, com balanceamento automático entre instâncias (caso existam múltiplas).

Você pode visualizar todos os serviços registrados acessando:

- http://localhost:8761

Benefícios gerados

- Documentação clara e isolada por serviço, facilitando manutenção e testes por equipe ou módulo.
- Roteamento automático via Eureka, eliminando a necessidade de configuração manual de rotas.
- Escalabilidade horizontal fácil com múltiplas instâncias por microserviço.

Repositório do Projeto

Instruções sobre como executar o projeto estão disponíveis no README do repositório no GitHub. O repositório pode ser acessado em:

https://github.com/angelovlima/beauty-book

Tecnologias e Ferramentas Utilizadas

O projeto utiliza um conjunto moderno e robusto de tecnologias para garantir modularidade, escalabilidade, testabilidade e qualidade de software:

- **Java 17**
  Linguagem principal utilizada no desenvolvimento de todos os microserviços, aproveitando recursos modernos como records, sealed classes, e melhorias de performance e segurança.
- **Spring Boot 3.2.4**
  Framework principal responsável pela configuração automática da aplicação, injeção de dependências, controle dos ciclos de vida e exposição de APIs REST.
- **Spring Web**
  Responsável por expor os endpoints REST, manipular requisições HTTP, respostas e configurar os controladores da camada API.

- **Spring Data JPA**
  Gerencia a camada de persistência com abstrações para repositórios e integração com o banco de dados PostgreSQL via JPA.
- **Spring Cloud Netflix Eureka Client**
  Permite o registro e a descoberta de serviços na arquitetura de microserviços via Eureka Server.
- **Spring Cloud OpenFeign**
  Utilizado para facilitar a comunicação entre microserviços por meio de clientes HTTP declarativos, promovendo desacoplamento e simplicidade.
- **Spring Boot DevTools**
  Usado durante o desenvolvimento local para hot reload de mudanças no código sem reinicialização manual do servidor.
- **PostgreSQL Driver**
  Driver de conexão com o banco de dados PostgreSQL em tempo de execução, utilizado por todos os microserviços.
- **Liquibase Core**
  Gerencia e versiona as migrations do banco de dados, garantindo consistência entre ambientes (dev, test e produção).
- **Lombok**
  Reduz a verbosidade do código com anotações como @Getter, @Builder, @RequiredArgsConstructor, evitando a escrita manual de métodos boilerplate.
- **SpringDoc OpenAPI**
  Gera automaticamente a documentação interativa da API com Swagger UI a partir das anotações dos controllers.
- **Maven**
  Ferramenta de build e gerenciamento de dependências dos projetos, além de facilitar o uso de plugins como o Jacoco.
- **Docker**
  Utilizado para containerizar os serviços e bancos de dados, garantindo reprodutibilidade, isolamento e facilidade de deploy.
- **H2 Database (somente em testes)**
  Banco de dados em memória usado para execução de testes de integração mais rápidos e isolados, sem impactar o PostgreSQL real.
- **JUnit 5**
  Framework de testes unitários e de integração, usado com alta cobertura, principalmente nos casos de uso e controladores.
- **RestAssured**
  Ferramenta para testes automatizados de APIs REST, especialmente útil nos testes de integração com validações de status e corpo de resposta.
- **Cucumber**
  Utilizado para testes BDD (Behaviour Driven Development), permitindo a escrita de testes em linguagem natural (.feature) e integração com o **JUnit.**

- **Testcontainers**
  Usado para testes de integração que dependem de serviços externos como banco de dados PostgreSQL ou MockServer, fornecendo containers isolados e temporários via Docker.
- **MockServer + MockServer Client Java**
  Utilizado para simular serviços externos durante testes de integração, especialmente útil para validar os gateways Feign com comportamento controlado.
- **JaCoCo Maven Plugin**
  Ferramenta de análise de cobertura de testes, utilizada para gerar relatórios e garantir que as regras de negócio estão bem testadas.

Deploy

A aplicação está preparada para múltiplos ambientes, com foco em automação e facilidade de manutenção:

- **Ambiente Local com Docker Compose**
  Utilizado para desenvolvimento e testes locais. Permite levantar todos os microserviços e banco de dados com um único comando.
- **VPS Hostinger com Ubuntu 24.04 (produção atual)**
  O deploy principal do projeto está configurado em uma VPS Linux (KVM 2 - Ubuntu 24.04) fornecida pela Hostinger, com acesso via SSH. O processo de deploy é **automatizado via GitHub Actions**:
    - A cada push na branch main, um workflow é acionado.
    - Todos os testes automatizados dos microserviços são executados
    - O código é enviado via SSH para a VPS utilizando o action easingthemes/ssh-deploy@v2.
    - As variáveis sensíveis (como chave SSH, host, porta, usuário e diretório de destino) estão protegidas como **secrets** no GitHub.

Esse fluxo garante **CI/CD contínuo**, com atualização automática do ambiente de produção a cada nova versão publicada no repositório.

Qualidade de Software e Testes

• Testes unitários com JUnit e Mockito
• Testes integrados com Testcontainers e PostgreSQL
• Testes BDD com Cucumber e RestAssured
• Testes de exceção e cobertura de falhas
• Validação de logs com Log Tracker
• Análise de cobertura com JaCoCo

Arquitetura Aplicada: Clean Architecture

O sistema foi inteiramente estruturado segundo os princípios da Clean Architecture, garantindo separação de responsabilidades, testabilidade, modularidade e independência de frameworks. Cada microserviço adota um padrão uniforme de camadas, facilitando a manutenção e a evolução contínua do sistema.

Estrutura de Camadas

A estrutura padrão de cada microserviço é composta pelas seguintes camadas:

domain.model

Contém as entidades centrais e seus value objects. Os modelos são imutáveis, encapsulados e validam regras mínimas de domínio. Exemplo: Customer, Booking, Cpf, Email.

domain.usecase

Agrupa os casos de uso da aplicação — como CreateCustomerUseCase, RebookAppointmentUseCase, CancelBookingUseCase. Cada caso de uso é implementado diretamente com @Service, favorecendo:

Injeção direta no controller sem criar camadas de serviços artificiais;

Clareza e rastreabilidade no fluxo da lógica;

Alta coesão e baixo acoplamento, pois o caso de uso conhece apenas interfaces da camada gateway.

domain.gateway

Define contratos de comunicação com qualquer sistema externo (banco, APIs, mensageria, etc). Exemplo: CustomerGateway, EstablishmentGateway. Essa abstração garante que a lógica de negócio nunca dependa de detalhes de infraestrutura.

api

Responsável por expor endpoints REST. Os controladores (ex: CustomerController, BookingController) apenas recebem as requisições, mapeiam DTOs e delegam os dados para os use cases.

Essa camada também é responsável pelas classes dto, mapper e configuração do Swagger, mantendo responsabilidades bem isoladas.

infra.gateway.jpa

Implementações concretas das interfaces da camada de domínio usando JPA. Os mappers convertem os objetos do domínio em entidades persistíveis (JpaEntity) e vice-versa, mantendo o domínio desacoplado da estrutura do banco.

infra.gateway.integration

Integrações com serviços externos (outros microserviços) usando Feign Clients. A arquitetura isola os clients das regras de negócio:

Os clients são interfaces com anotações Feign.

Os gateways de integração são componentes intermediários responsáveis por tratar erros (como FeignException.NotFound) e mapear dados.

A camada de domínio consome apenas a interface EstablishmentGateway, por exemplo, sem saber que há um Feign Client por trás.

Exemplo Prático: Booking + Integração com Estabelecimento

O EstablishmentClient (infra.integration.client) define a comunicação com o microserviço de gerenciamento de estabelecimentos via Feign.

O EstablishmentIntegrationGateway encapsula essa comunicação, trata exceções específicas e realiza o mapeamento com ServiceOfferedDtoMapper.

A interface EstablishmentGateway no domínio é usada pelos casos de uso para consultar horários de funcionamento, buscar dados do estabelecimento e duração de serviços.

O domínio não depende de Feign, nem conhece DTOs HTTP — ele apenas define o comportamento esperado.

Qualidade Arquitetural Evidenciada

Abstração por contrato: o domínio conhece apenas interfaces; infraestrutura injeta implementações concretas.

Independência tecnológica: possível substituir JPA por MongoDB ou Feign por RESTTemplate sem afetar as regras de negócio.

Alto isolamento entre camadas: mapeamentos entre DTOs, entidades JPA e objetos de domínio são realizados em mappers dedicados.

Facilidade de testes: use cases são unitários, testáveis com mocks dos gateways. Controladores e integrações podem ser testados com Testcontainers.

Tratamento robusto de falhas: falhas como FeignException.NotFound são convertidas em exceções de domínio (EstablishmentNotFoundException).

Padrão consistente entre microserviços, mantendo legibilidade e organização do código.