Relatório do Trabalho - Seu Cantinho

Disciplina: ci1163 – Design de Software
Alunos: Bruno Vila Lobus Strapasson e Diego Vieira de Christo
GRR: 20215522 e 20206154

1. Introdução

Este documento apresenta a arquitetura e o projeto de software do sistema **Seu Cantinho**, uma plataforma para gerenciamento de espaços para eventos e reservas. O sistema foi desenvolvido utilizando a linguagem **Java 17**, com o framework **Spring Boot**, seguindo uma arquitetura **em camadas (Controller → Service → Repository → Model)**, complementada por práticas de orientação a objetos e mapeamento JPA/Hibernate.

2. Estilo Arquitetural Escolhido: Arquitetura em Camadas

A arquitetura adotada no projeto foi a **arquitetura em camadas**, por ser a mais adequada aos requisitos de qualidade definidos para o sistema *Seu Cantinho*. Essa escolha foi fundamentada na necessidade de **simplicidade, modularidade, manutenibilidade, escalabilidade moderada e confiabilidade**, conforme exigido no desenvolvimento do protótipo.

1. Simplicidade

A arquitetura em camadas facilita o aprendizado da arquitetura por novos desenvolvedores e reduz complexidade. Deixando o desenvolvimento mais rápido e reduzindo os erros.
Ela segue uma estrutura já conhecida:

- **Controller** → interface com o cliente/REST
- **Service** → regras de negócio
- **Repository** → acesso a dados
- **Model/Entity** → representação do domínio
- **DTOs** → transporte de dados

2. Modularidade e Separação de Responsabilidades

A divisão por camadas garante que cada parte do código tenha um papel único:

- **Controllers** lidam com requisições HTTP.
- **Services** centralizam a lógica.
- **Repositories** encapsulam o acesso ao banco.
- **Models** representam o domínio.

Essa modularidade **evita acoplamento** entre camadas e permite evolução de componentes de forma isolada.

Vantagens:

- Alterar o banco não afeta controllers nem regras de negócio.
- Atualizar a lógica de Pagamentos ou Reservas não quebra a API.
- Facilita testes unitários: cada camada pode ser testada separadamente.

3. Manutenibilidade

Manutenibilidade foi um critério essencial para o projeto, pois a solução deve ser fácil de compreender, corrigir e evoluir.

A arquitetura em camadas oferece:

- Código organizado e padronizado
- Fluxo previsível (controller → service → repo)
- Facilidade de localizar bugs ⇢ basta identificar a camada afetada

4. Escalabilidade

Embora REST em camadas não seja a arquitetura mais escalável de todas (como microserviços), ela oferece **escalabilidade suficiente** para um protótipo, com possibilidade de evolução futura que é o que está sendo pedido para esse momento.

A presença de services e repositories bem separados permite que futuramente:

- Serviços de alta demanda (ex: reservas) sejam migrados para microserviços independentes.
- Caches possam ser adicionados na camada de serviço.
- Balanceamento de carga possa ser colocado acima da camada controller.

5. Confiabilidade

A confiabilidade aumenta quando o sistema está organizado de forma clara.
A divisão em camadas permite:

- Tratamento consistente de erros
- Validações centralizadas
- Regras de negócio não duplicadas

Na implementação adotada do sistema Seu Cantinho, temo que:

- Todo pagamento passa pelo *PagamentoService*
- Toda reserva passa pelo *ReservaService*

Dessa forma evita inconsistências de regras e aumenta segurança do estado do sistema

Trade-offs da Arquitetura em Camadas

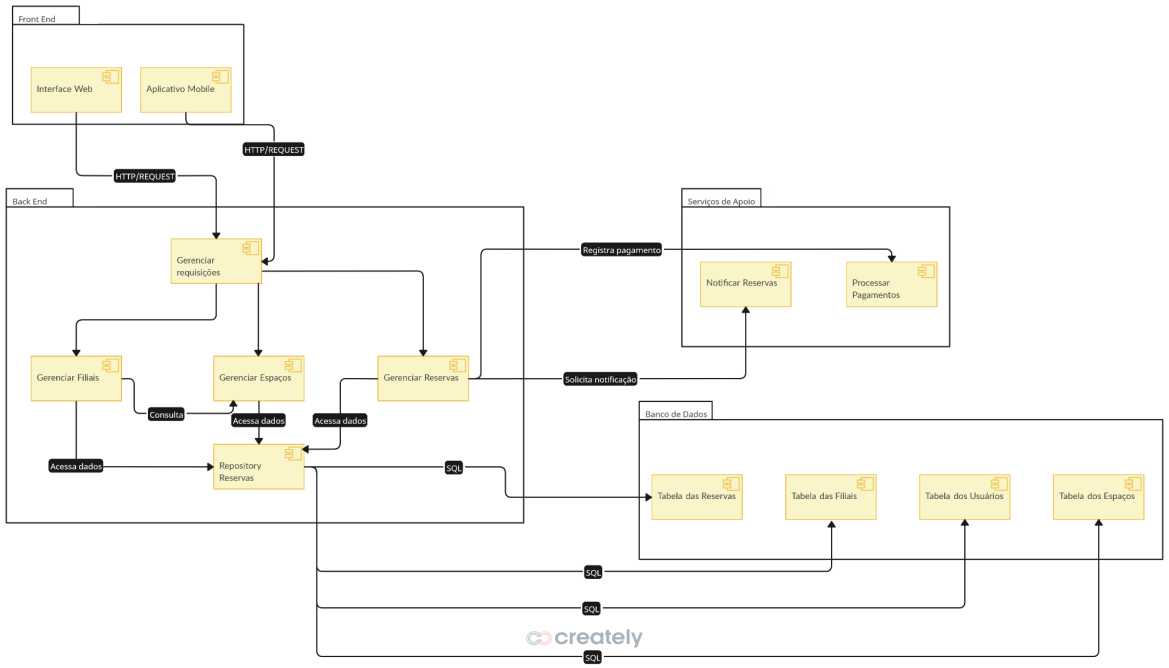
Vantagens

- Separação clara de responsabilidades
- Facilidade de manutenção
- Baixo acoplamento e alta coesão
- Curva de aprendizado pequena
- Organização natural para projetos CRUD

Desvantagens

- ✘ Possível **overhead** ao atravessar várias camadas para operações simples
- ✘ Risco de **domínio anêmico** quando as entidades viram apenas “sacos de dados”
- ✘ Pode exigir infraestrutura mais complexa quando o sistema cresce muito
- ✘ Nem sempre é ideal para operações altamente performáticas (ex: consultas massivas)

3. Diagrama de Componentes

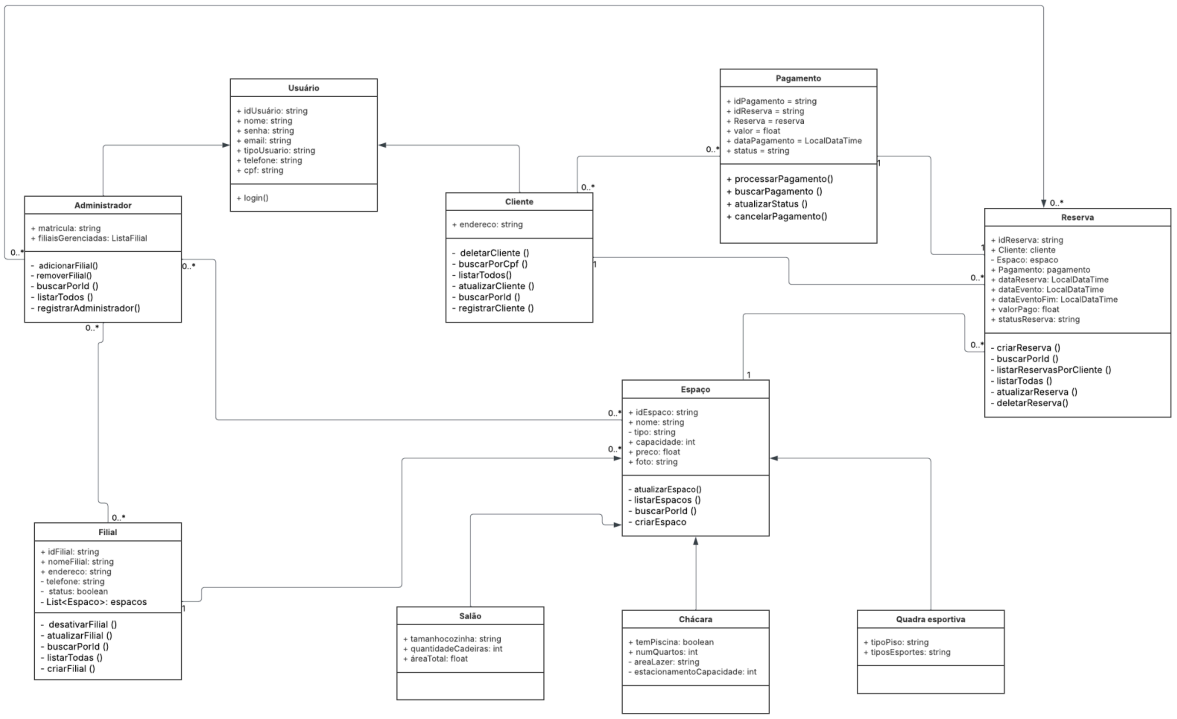


Explicação:

- **Front-End** (web/mobile) consumindo a API REST.
- **Backend Spring Boot** dividido em componentes lógicos:
    - Gerenciamento de Filiais
    - Gerenciamento de Espaços
    - Gerenciamento de Reservas
    - Pagamentos
    - Usuários
- **Camada de Persistência** (JPA Repositories)
- **Banco de Dados Relacional**

A comunicação entre componentes ocorre via chamadas internas (Service ↔ Repository), enquanto a comunicação externa é via HTTP.

4. Diagrama de Classes



O diagrama representa todo o modelo de domínio do sistema.
Inclui:

- Usuário (abstrata)
    - Cliente
    - Administrador
- Filial
- Espaco (abstrata)
    - Salao
    - Chacara
    - QuadraEsportiva
- Reserva
- Pagamento

5. Principais decisões:

▸ Herança para modelar tipos diferentes de espaço

Isso evita duplicação e mantém atributos comuns em *Espaco*.

▸ Relacionamentos essenciais modelados como:

- Cliente 1—0..* Reserva
- Filial 1—N Espaço
- Espaço 1—0..* Reserva
- Reserva 1—1 Pagamento
- Administrador N—N Filial

▸ Uso de JOINED em herança (JPA)

Evita duplicação e facilita consultas.

O mapeamento UML → código

UML → Java (Model)

Classe UML	Classe Java
Usuario	Usuario.java
Cliente	Cliente.java
Administrador	Administrador.java
Espaco	Espaco.java
Salao	Salao.java

Chacara	Chacara.java
QuadraEsportiva	QuadraEsportiva.java
Reserva	Reserva.java
Pagamento	Pagamento.java
Filial	Filial.java

UML → Relacionamentos

- Reserva —1:1— Pagamento → @OneToOne
- Filial —1:N— Espaço → @OneToMany
- Administrador N—N Filial → @ManyToMany

6. Regras de Negócio e Decisões de Implementação

6.1 Regras de Negócio Implementadas nos Services

AdministradorService

- Apenas administradores podem cadastrar, editar ou remover espaços.
- O sistema impede a criação de usuários administradores com e-mails duplicados.
- Validação de senha mínima e formato de e-mail.
- Administradores têm acesso às reservas para fins de auditoria.

ClienteService

- Cadastro de clientes exige e-mail único.

- Um cliente só pode cancelar as próprias reservas.
- Clientes não podem criar reservas para períodos já ocupados.
- Impede exclusão de cliente caso ele tenha reservas futuras (regra de integridade).

EspacoService

- Um espaço só pode ser ativado se possuir informações mínimas (nome, endereço, capacidade).
- Não permite exclusão física de espaços que já possuem reservas — apenas *desativação lógica* (ativo = false).
- Atualizações de preço ou capacidade não afetam reservas já realizadas.
- Espaços desativados não aparecem em listagens públicas.

ReservaService

- O cliente só pode reservar se:
    - O espaço estiver ativo.
    - Não houver conflito de horário com outra reserva confirmada.
- Proibição de reservas retroativas (datas no passado).
- Cálculo automático do valor da reserva baseado no valor/dia do espaço.
- Cancelamento permitido apenas até X horas antes do início (ajustável no código).
- Regras de conflito de datas:
    - dataInicioNova < dataFimExistente **e**
    - dataFimNova > dataInicioExistente → conflito.

6.2 Considerações sobre Validação de Dados

- **E-mail** validado por regex em todos os cadastros.
- **Datas** validadas para garantir:

    - Início < Fim
    - Datas futuras
    - Reservas não sobrepostas
- **Campos obrigatórios** tratados com:
    - @NotNull, @NotBlank, @Size
    - Validação adicional nas Services quando necessário.
- **IDs inexistentes** retornam erros padrão 404 – Not Found.
- **Banco de Dados** reforça integridade com:
    - chaves estrangeiras para cliente, espaço e reservas;
    - cascades controlados para evitar exclusões indevidas.

7. Limitações Conhecidas do Protótipo

- **Sistema ainda não possui autenticação real**
  (usuário não loga com token; autorização é simulada no backend).
- **Não há controle de concorrência otimista**
  → Reservas simultâneas podem disputar o mesmo horário em ambientes de alta carga.
- **Sem envio automático de e-mails**
  (confirmação de reserva é apenas lógica).
- **Cancelamento de reserva sem multa**
  → o protótipo não implementa regras financeiras.
- **Falha de idempotência**
  → reenviar a mesma requisição de reserva pode duplicar ações.

8. Reflexões Finais

O desenvolvimento do sistema *Seu Cantinho* permitiu aplicar na prática diversos conceitos de design de software ensinados ao longo da disciplina. A escolha da arquitetura em camadas mostrou-se adequada para o escopo do projeto, pois ofereceu simplicidade, organização e modularização suficientes para um sistema de CRUD com regras de negócio

bem definidas. Essa arquitetura facilitou a evolução incremental do código, especialmente conforme novas operações foram adicionadas ao código sem que isso impactasse o restante da aplicação.

A modelagem UML ajudou com quais classes seriam criadas, como seria o relacionamento entre elas, já fazendo um esboço de como seria o fluxo permitindo onde cada camada poderia atuar. Além de definir as heranças entre as classes. O diagrama de componentes foi feito no creately e o de classes no lucidchart.

Durante a implementação, foi muito importante a separação de responsabilidades: controllers focaram na API REST, services centralizaram regras de negócio e os repositories encapsularam o acesso ao banco. Essa divisão contribuiu para um código mais limpo, com baixo acoplamento e facilitando os testes para cada nova funcionalidade.

Por outro lado, também foi possível observar alguns trade-offs naturais desse estilo arquitetural. Em certos momentos, especialmente em operações simples, a navegação entre múltiplas camadas introduz um leve overhead.

O resultado final é um protótipo funcional, bem estruturado e pronto para evolução futura, cumprindo todos os requisitos estabelecidos na especificação.