Sistema de Gestão de Reservas de Espaços “Seu Cantinho” Planejamento de Arquitetura, Modelagem e Implementação

Vitor Faria, João Meyer

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

vfms23@inf.ufpr.br, jmm23@inf.ufpr.br

Repositório GitHub: https://github.com/jmeyerr12/seu-cantinho

1. Introdução

O “Seu Cantinho” é uma rede de locação de espaços para festas e eventos (salões, chácaras, quadras esportivas etc.). O sistema atual, criado de forma emergencial, já não atende às necessidades de escala e confiabilidade em múltiplas unidades, expondo riscos de *double booking*, perda de informações financeiras e dificuldades de uso.

Este documento descreve o plano para evoluir o sistema a uma solução integrada, confiável e de operação simples, com foco em:

- cadastro de espaços com fotos, capacidade e preço;
- gestão de reservas por cliente, data e valor;
- indicação clara do status de pagamento (sinal/quitado).

Também são delineadas modelagem UML, decisões arquiteturais e execução via conteinerização.

2. Objetivos

Os objetivos que orientam o desenvolvimento são:

1. garantir consistência de reservas entre unidades;
2. reduzir erros operacionais (conflitos de horário);
3. oferecer interface clara e tolerante a falhas;
4. centralizar espaços, reservas e pagamentos em base única.

Do ponto de vista técnico, o sistema:

- expõe funcionalidades por APIs REST documentadas (OpenAPI/Swagger);
- adota arquitetura em camadas, extensível;
- suporta execução integrada via Docker Compose;
- mantém código, diagramas e documentação versionados em Git.

3. Arquitetura do Sistema

Adota-se uma arquitetura monolítica em camadas. As funcionalidades de usuários, espaços, reservas e pagamentos residem em um único serviço lógico. A organização interna segue o fluxo rotas → middlewares → controllers → services → persistência.

Os *controllers* passaram a ser intencionalmente “finos”: recebem a requisição HTTP, fazem validações básicas de entrada/saída e delegam a lógica de negócio para módulos de *services*. Essa camada de serviços concentra regras como validação de conflitos de horário, cálculo de valores de reserva, regras de cancelamento e políticas de autenticação, reduzindo o acoplamento com o Express e facilitando testes unitários.

3.1. Camada de Persistência (Banco de Dados)

O PostgreSQL é o SGBD relacional de base. Estão previstas:

- tabelas de usuários e autenticação;
- tabela de espaços (nome, descrição, capacidade, preço e chaves de imagens);
- tabela de reservas (cliente, data/horário, espaço, valor);
- campos/estruturas para status de pagamento (sinal/quitado).

As fotos não são armazenadas como BLOB; apenas referências (chaves) ficam no banco.

3.2. Camada de Armazenamento de Imagens (MinIO)

O armazenamento de fotos utiliza MinIO (compatível com S3). O banco guarda somente as chaves/URLs internas das imagens. Com isso, obtém-se:

- reaproveitamento de imagens em diferentes páginas;
- uploads/consultas de arquivos grandes sem onerar o tráfego transacional;
- gestão centralizada dos ativos visuais dos espaços.

3.3. Camada de Backend

O backend é desenvolvido em TypeScript com Express. As funcionalidades são expostas por APIs REST através de um ponto central de rotas, que direciona cada requisição para o *controller* correspondente.

O fluxo interno segue a ordem:

1. rotas registradas em *app.ts*;
2. aplicação dos middlewares de autenticação (JWT) e permissões quando necessário;
3. chamada dos *controllers*;
4. delegação para *services* de domínio (usuários, espaços, reservas, pagamentos);
5. acesso ao banco de dados e ao armazenamento de imagens.

Cada módulo de serviço (`branchService`, `spaceService`, `reservationService`, `paymentService`, `userService` etc.) encapsula a lógica específica daquele contexto, incluindo as consultas SQL necessárias. Assim, a regra de negócio não fica espalhada por múltiplos endpoints, mas centralizada em funções reutilizáveis e testáveis.

As rotas de `/users/login` e `/users/signup` não usam JWT, pois é o próprio login que gera o token. As validações de payload são mantidas em nível de controller e service, com checagens mínimas antes de acessar o banco.

3.4. Camada de Frontend

O frontend utiliza Next.js (React) e consome as APIs REST. Entre as responsabilidades:

- telas de cadastro/edição de espaços;
- gestão de reservas (inclusão, visualização e edição);
- exibição do status de pagamento;
- exibição de fotos a partir das chaves do MinIO;
- fluxos de login e acesso autenticado.

A organização por páginas e o suporte a SSR permitem evoluções futuras de SEO e desempenho.

4. Modelagem UML

Serão mantidos:

- Diagrama de Classes: Usuário, Espaço, Reserva e Pagamento, com seus relacionamentos (p. ex., Reserva associada a um Espaço e a um Cliente).
- Diagrama de Componentes/Arquitetura em Camadas: separação entre frontend (Next.js), backend (Express/TypeScript), banco (PostgreSQL) e armazenamento (MinIO).

5. APIs REST

A comunicação entre frontend e backend ocorre via APIs REST, com grupos de endpoints como:

- `/users`: criação e autenticação de usuários;
- `/spaces`: cadastro, listagem e edição de espaços (metadados e chaves de imagens);
- `/reservations`: criação e consulta de reservas (cliente, data/horário, espaço);
- `/payments`: consulta do status de pagamento (sinal/quitada).

As APIs são documentadas em OpenAPI, facilitando testes por `curl` ou Swagger UI. A camada de serviços garante que regras de negócio (como prevenção de conflitos de horário) sejam aplicadas de forma consistente, independentemente de qual endpoint esteja sendo chamado.

6. Execução via Docker Compose

A execução integrada utiliza Docker Compose para orquestrar PostgreSQL, backend (Node/Express), frontend (Next.js) e MinIO. Entre os ganhos práticos:

- ambiente de desenvolvimento reprodutível;
- inicialização simplificada para demonstrações/avaliações;
- menor atrito com configuração manual de portas, variáveis e dependências.

6.1. Criação de usuário administrador

Para fins de testes e demonstrações, um usuário administrador pode ser criado manualmente após o registro normal pela interface. O fluxo é:

1. Registre o usuário normalmente pela plataforma (fluxo padrão de cadastro).
2. Acesse o container do PostgreSQL:
   ```
   docker exec -it seu_cantinho_db psql -U seucantinho -d seucantinho
   ```
3. Atualize o campo `role` do usuário recém-criado para o papel de administrador, por exemplo:
   ```
   UPDATE users SET role = 'admin' WHERE email = 'email_do_usuario';
   ```

O usuário administrador possui permissões ampliadas, incluindo:

- criação de filiais;
- aprovação de pagamentos;
- cadastro e edição de espaços;
- upload e gerenciamento de fotos dos espaços.

Esse procedimento facilita a avaliação do sistema sem expor rotas de criação de administradores diretamente pela API.

7. Decisões de Projeto

As decisões que sustentam o plano são:

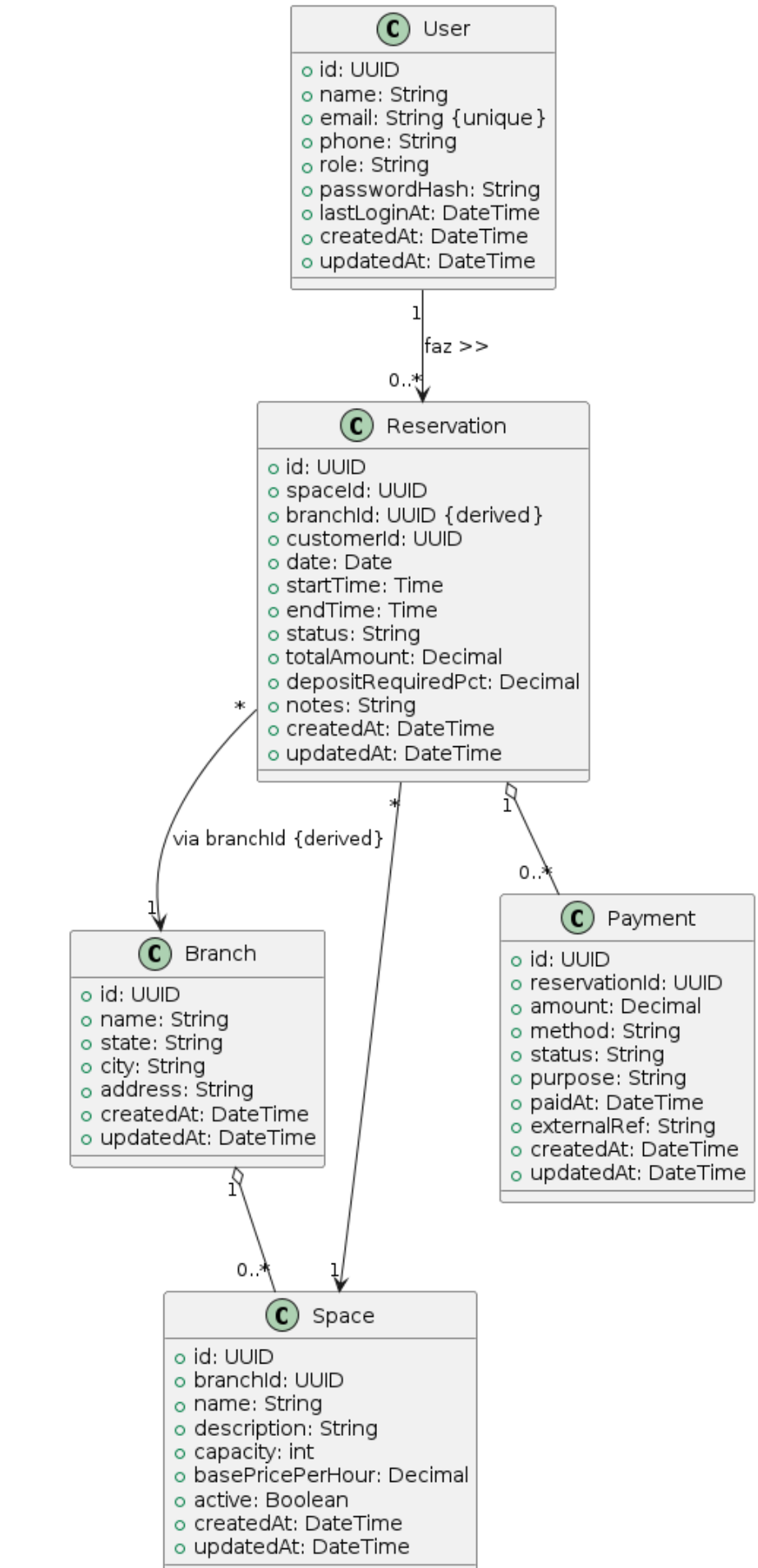
- Monólito em camadas: simplicidade de desenvolvimento e implantação. Toda a lógica está em um único serviço, organizado em rotas, middlewares (JWT e permissões), controllers e *services* com acesso ao banco de dados e ao MinIO.
- Camada de serviços de domínio: controllers finos delegam a lógica para módulos de serviço especializados, que concentram as regras de negócio, a orquestração de consultas SQL e os cálculos específicos (por exemplo, valor total de uma reserva). Isso facilita a reutilização de regras entre diferentes endpoints e melhora a testabilidade do código.
- Prevenção de *double booking*: toda criação ou alteração de reserva passa pela camada de serviços (`reservationService`), que executa uma consulta de conflito antes de persistir os dados. Essa consulta verifica, para o mesmo espaço e data, se já existe alguma reserva não cancelada cujo intervalo de horário se sobreponha ao intervalo solicitado, usando a condição:

  $$\neg(\text{end\_time} \leq \text{novo\_início} \ \vee \ \text{start\_time} \geq \text{novo\_fim}).$$

  Na prática, isso significa que:
  - ao criar uma reserva, o serviço busca um registro conflitante e, se encontrar, devolve erro HTTP 409 (*conflict*) em vez de inserir a linha;
  - ao atualizar data/horário de uma reserva existente, o mesmo critério é reavaliado, desconsiderando a própria reserva (via filtro por `id`) e impedindo que a alteração crie um novo conflito;
  - como toda lógica de reserva passa por esse serviço, qualquer nova rota ou caso de uso reutiliza a mesma regra centralizada, evitando implementações "paralelas" suscetíveis a *bugs*.

  Essa combinação de verificação de sobreposição na camada de serviços e uso de um único ponto de entrada para operações de reserva materializa, na arquitetura, o requisito de evitar *double booking* entre unidades e horários.
- Autenticação JWT: apenas rotas protegidas exigem o token; `/users/login` e `/users/signup` são públicas e o login gera o JWT.
- PostgreSQL: integridade e recursos adequados para evitar conflitos de reserva;
- Express + TypeScript: tipagem estática e ecossistema maduro para APIs REST;
- Next.js: desenvolvimento ágil de páginas e possibilidade de SSR;
- MinIO: mídia fora do banco transacional, com escalabilidade e backup simplificado.

Seu Cantinho — Diagrama de Classes (Domínio) — Sem Enums (limpo)



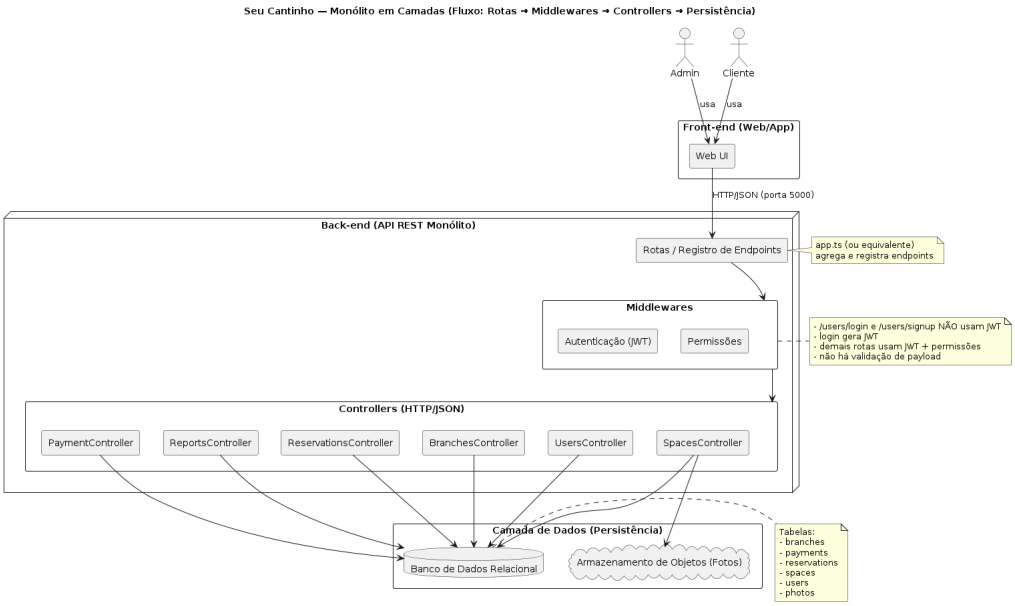


Figura 2. Diagrama de Componentes.

Figura 3. Árvore de utilidades.