Documentação do Sistema *Meu Cantinho*

Arquitetura, Mapeamento UML–Código, Decisões de Design e Execução

Thiago Zilio – GRR20234265

Tiago Augusto – GRR20232334

Disciplina de Projeto de Design de Software

Universidade Federal do Paraná

1 de dezembro de 2025

Repositório do projeto no GitHub:
`https://github.com/tzilio/meu-cantinho`

Sumário

1 Introdução **3**

2 Arquitetura Proposta **3**

2.1 Visão Geral em Três Camadas 3

2.2 Arquitetura do Backend 4

2.3 Arquitetura do Banco de Dados 5

2.4 Arquitetura do Frontend 5

3 Mapeamento UML → Código **6**

3.1 Artefatos UML 6

3.2 Classes de Domínio → Tabelas 7

3.3 Classes UML → Código Backend 7

3.4 Classes UML → Código Frontend 8

4 Decisões de Design e Trade-offs **8**

4.1 Separação Backend/Frontend e Uso de Swagger 8

4.2 Validação de Regras de Negócio em Dois Níveis 8

4.3 Modelo de Pagamentos Múltiplos e Recomputação de Status 9

4.4 Upload de Capa de Espaço 9

5 Instruções de Execução **9**
5.1 Execução com Docker Compose 9
5.2 Execução Local (sem Docker) 10

6 Considerações Finais **11**

1 Introdução

Este relatório documenta o sistema *Meu Cantinho*, desenvolvido ao longo da disciplina de Projeto de Design de Software. O sistema oferece uma solução para gerenciamento de filiais, espaços de locação, clientes, reservas e pagamentos, com suporte a múltiplos ambientes (desenvolvimento e produção via Docker).

A documentação está organizada de modo a cobrir quatro eixos principais:

- a **arquitetura proposta**, descrevendo como backend, frontend e banco de dados se relacionam, quais responsabilidades cada camada assume e como os componentes internos (controllers, serviços, rotas, views) são estruturados;
- o **mapeamento UML → código**, mostrando como as classes e relacionamentos modelados nos diagramas foram traduzidos para tabelas do banco, interfaces TypeScript e módulos concretos em TypeScript/JavaScript;
- as **decisões de design** tomadas durante o projeto, como a separação explícita entre frontend e backend, a modelagem de pagamentos múltiplos por reserva e a presença de serviços auxiliares (por exemplo, para upload de capas de espaços);
- os **trade-offs identificados**, discutindo os custos e benefícios de certas escolhas, como duplicar validações no frontend e no backend para melhorar a experiência do usuário ao custo de manter regras em dois lugares.

Ao detalhar a arquitetura, o relatório explicita não apenas a estrutura física dos diretórios (`backend/`, `frontend/`, `database/`, `uml/`), mas também os fluxos de dados típicos: do formulário nas views (Vue + Vuetify), passando pelos controllers Express, até as consultas SQL no PostgreSQL. No mapeamento UML–código, busca-se evidenciar que cada entidade central do domínio (`Branch`, `Space`, `Customer`, `Reservation`, `Payment`) possui uma contraparte clara no schema e nas interfaces/tipos usados pelo frontend.

Já na discussão de design e trade-offs, o foco está em justificar as opções tomadas à luz dos objetivos do projeto: simplicidade de implementação, clareza arquitetural e flexibilidade para futuras extensões (por exemplo, integração com gateways de pagamento reais ou mecanismos de autenticação mais complexos). Dessa forma, o relatório não apenas descreve o sistema atual, mas também registra o raciocínio por trás da solução adotada.

2 Arquitetura Proposta

2.1 Visão Geral em Três Camadas

A arquitetura geral segue o padrão clássico de três camadas:

- **Camada de Apresentação**: diretório `frontend/`, com aplicação Vue 3, Vuetify e Vite, responsável pelas telas de cadastros e consultas.
- **Camada de Lógica de Negócio**: diretório `backend/`, com API REST em Node.js/Express e TypeScript.
- **Camada de Dados**: diretório `database/`, com o script `init.sql` para criação do schema no PostgreSQL.

O orquestrador `docker-compose.yml` integra esses componentes, definindo serviços para backend, frontend e banco de dados.

2.2 Arquitetura do Backend

O backend encontra-se em `backend/` e é organizado da seguinte forma:

- `src/main.ts`: ponto de entrada da aplicação; instancia o servidor HTTP importando `app` e definindo a porta padrão.
- `src/app.ts`: construção da aplicação Express; registra middlewares globais (JSON, CORS, *logging*) e o roteador principal.
- `src/server.ts`: variante de inicialização que separa configuração da app da subida efetiva do servidor.
- `src/db.ts`: configuração da conexão com o PostgreSQL via `pg.Pool`, utilizado por todos os controllers para executar queries.
- `src/routes/index.ts`: define as rotas REST principais (`/branches`, `/spaces`, `/customers`, `/reservations`, `/payments`), associando-as aos respectivos controllers.
- `src/routes/uploads.ts`: rotas específicas para upload de arquivos (por exemplo, imagens de capa dos espaços).
- `src/controllers/branch.ts`: operações CRUD de filiais (`branches`), como criação, listagem, atualização e remoção.
- `src/controllers/space.ts`: operações sobre espaços (`spaces`), incluindo capacidade, preço por hora e associação com filiais.
- `src/controllers/customer.ts`: CRUD de clientes (`customers`).
- `src/controllers/reservation.ts`: criação, listagem e cancelamento de reservas; validação de intervalo de datas/horários e cálculo do `total_amount`.

- `src/controllers/payment.ts`: registro de pagamentos, confirmação de pagamento, listagem filtrada e regras de não exceder o saldo restante da reserva.
- `src/middleware/auth.ts`: middleware de autenticação (quando aplicável), responsável por validar tokens ou sessões antes de permitir acesso a determinadas rotas.
- `src/services/spaceCoverUpload.ts`: serviço específico para upload de imagens de capa de espaços, encapsulando detalhes de armazenamento (por exemplo, diretórios locais ou S3).
- `src/swagger.ts`: configuração da documentação da API via Swagger/OpenAPI, permitindo explorar os endpoints, parâmetros e respostas através de uma UI interativa.

O `Dockerfile` do backend define a imagem Node, copia os fontes, instala dependências e expõe a porta da API.

2.3 Arquitetura do Banco de Dados

O schema do banco está em `database/init.sql`. Os principais objetos são:

- `branches`: filiais (`id`, `name`, `state`, `city`, `address`).
- `spaces`: espaços vinculados a uma filial (`branch_id`), com `capacity`, `price_per_hour`, `cover_url` e indicador de `active`.
- `customers`: clientes com `name`, `email` (único) e `phone`.
- `reservations`: reservas com `check_in_date`, `check_out_date`, `start_time`, `end_time`, `total_amount`, `deposit_pct`, `status` e `adults_count`.
- `payments`: pagamentos vinculados a uma reserva (`reservation_id`), contendo `amount`, `method`, `status`, `purpose`, `paid_at` e `external_ref`.

Constraints de integridade referencial (`FOREIGN KEY`), de domínio (`CHECK`) e índices otimizados (`idx_spaces_branch_active`, `idx_reservations_space_checkin`, etc.) dão suporte às queries mais frequentes (por exemplo, listar reservas por espaço e período).

2.4 Arquitetura do Frontend

O frontend está em `frontend/` e organizado da seguinte maneira:

- `src/main.ts`: ponto de entrada que monta a aplicação Vue, registra o roteador e o plugin do Vuetify.

- `src/App.vue`: componente raiz, com *layout* padrão (barra superior, conteúdo principal, etc.).
- `src/plugins/vuetify.js`: configuração do Vuetify (tema, componentes, ícones).
- `src/router/index.js` + `src/router.ts`: definição das rotas de SPA, apontando para as views de filiais, espaços, clientes, reservas e pagamentos.
- `src/services/http.ts`: client HTTP baseado em Axios, configurado com a base URL da API (via variáveis de ambiente Vite).
- `src/stores/branchesStore.ts`: store reativa para dados de filiais, utilizada em componentes que necessitam da lista global de branches.
- `src/views/BranchesView.vue`: tela de CRUD de filiais.
- `src/views/SpacesView.vue`: tela para cadastro e gerenciamento de espaços, incluindo upload da capa.
- `src/views/CustomersView.vue`: tela de cadastro de clientes.
- `src/views/ReservationsView.vue`: tela de criação e listagem de reservas, com validação de capacidade, conflitos de horário e exibição de valores calculados.
- `src/views/PaymentsView.vue`: tela de registro de pagamentos (à vista e parcelado), com filtros, status e integração com a lógica de saldo restante.
- `src/types.ts`: definições de tipos TypeScript (`Branch`, `Space`, `User/Customer`, `Reservation`, `Payment`), espelhando o schema e a resposta da API.

O `Dockerfile` do frontend constrói o bundle com Vite e serve a aplicação, normalmente atrás de um `node:alpine` ou similar.

3 Mapeamento UML → Código

3.1 Artefatos UML

Os diagramas UML foram produzidos e salvos no diretório `uml/`:

- `uml/classes.drawio`: diagrama de classes do domínio, representando entidades como `Branch`, `Space`, `Customer`, `Reservation` e `Payment`.
- `uml/componentes.drawio`: diagrama de componentes, mostrando a interação entre frontend, backend, banco de dados e serviço de upload de arquivos.

Opcionalmente, as figuras podem ser exportadas para PDF/PNG e incluídas no relatório com:

```
\begin{figure}[h]
  \centering
  \includegraphics[width=0.9\textwidth]{uml/classes.pdf}
  \caption{Diagrama de classes do sistema Meu Cantinho.}
\end{figure}
```

3.2 Classes de Domínio → Tabelas

O mapeamento das classes do diagrama de classes para o schema relacional é:

- **Branch** → tabela `branches`;
- **Space** → tabela `spaces`;
- **Customer** → tabela `customers`;
- **Reservation** → tabela `reservations`;
- **Payment** → tabela `payments`.

As associações UML tornam-se chaves estrangeiras:

- `spaces.branch_id` referencia `branches.id`;
- `reservations.space_id` referencia `spaces.id`;
- `reservations.branch_id` referencia `branches.id`;
- `reservations.customer_id` referencia `customers.id`;
- `payments.reservation_id` referencia `reservations.id`.

3.3 Classes UML → Código Backend

No backend, cada entidade de domínio tem um conjunto de operações associadas:

- **Branch**: operações em `src/controllers/branch.ts` implementam os casos de uso de administrar filiais.
- **Space**: `src/controllers/space.ts` e `src/services/spaceCoverUpload.ts` cuidam tanto de dados como do upload de capa.
- **Customer**: `src/controllers/customer.ts` implementa CRUD de clientes.

- **Reservation**: `src/controllers/reservation.ts` cuida de criação/listagem/cancelamento, além de checagens de sobreposição de horários e consistência de datas.
- **Payment**: `src/controllers/payment.ts` implementa o registro de pagamentos, validação de valores e confirmação, com `recomputeReservationStatus` recalculando o status da reserva.

3.4 Classes UML → Código Frontend

No frontend, as classes do domínio são refletidas nas interfaces TypeScript em `src/types.ts`:

- `Branch` (UML) → `interface Branch`;
- `Space` → `interface Space`;
- `Customer` → `interface User` (ou `Customer`, conforme definido);
- `Reservation` → `interface Reservation`;
- `Payment` → `interface Payment`.

Esses tipos são usados nas views (`BranchesView.vue`, `SpacesView.vue`, `ReservationsView.vue`, `PaymentsView.vue`) para tipar dados reativos, respostas da API e props de componentes de tabela.

4 Decisões de Design e Trade-offs

4.1 Separação Backend/Frontend e Uso de Swagger

A opção por separar backend e frontend em projetos distintos (`backend/` e `frontend/`) facilita a implantação independente e a evolução separada das camadas. A documentação via Swagger (`src/swagger.ts`) formaliza os endpoints da API e serve de contrato explícito entre as equipes (ou entre as camadas).

4.2 Validação de Regras de Negócio em Dois Níveis

Algumas regras de negócio são implementadas tanto no backend quanto no frontend, por exemplo:

- verificação de conflitos de horário de reservas;
- verificação de capacidade de adultos por espaço;
- verificação de que o valor do pagamento não excede o saldo restante.

Trade-off:

- **Pró**: melhor UX (feedback imediato) e integridade garantida no servidor;
- **Contra**: duplicidade de lógica e necessidade de manter as regras sincronizadas.

4.3 Modelo de Pagamentos Múltiplos e Recomputação de Status

A modelagem de `payments` como uma entidade separada, permitindo vários registros por reserva, torna o sistema apto a lidar com:

- pagamentos parciais (`purpose = DEPOSIT` / `BALANCE`);
- diferentes métodos por parcela (PIX, cartão, dinheiro, boleto);
- cancelamentos e reembolsos.

A função `recomputeReservationStatus` (em `payment.ts`) recalcula se a reserva deve ser marcada como `CONFIRMED` quando a soma dos pagamentos `PAID` atinge o `total_amount`.

4.4 Upload de Capa de Espaço

O serviço `spaceCoverUpload.ts` centraliza a lógica de upload de capa de espaços, isolando detalhes de filesystem ou serviços externos de storage das demais camadas.

5 Instruções de Execução

5.1 Execução com Docker Compose

Na raiz do projeto, onde se encontra `docker-compose.yml`, os passos são:

1. Construir e subir os serviços:

   ```
   docker compose up --build
   ```

2. O serviço de banco rodará com o script `database/init.sql` inicializando o schema.
3. O backend será exposto em uma porta como `http://localhost:3000` (conforme configuração).
4. O frontend será exposto em uma porta como `http://localhost:5173`.

5.2 Execução Local (sem Docker)

Banco de Dados

1. Criar o banco PostgreSQL:

   ```
   createdb meu_cantinho
   ```

2. Executar o script:

   ```
   psql -d meu_cantinho -f database/init.sql
   ```

Backend

1. Entrar em `backend/` e instalar dependências:

   ```
   cd backend
   npm install
   ```

2. Configurar o arquivo `.env` com a URL de conexão:

   ```
   DATABASE_URL=postgres://usuario:senha@localhost:5432/meu_cantinho
   ```

3. Rodar em modo desenvolvimento:

   ```
   npm run dev
   ```

Frontend

1. Entrar em `frontend/`:

   ```
   cd frontend
   npm install
   ```

2. Configurar `.env.development` com a base da API:

   ```
   VITE_API_BASE_URL=http://localhost:3000
   ```

3. Rodar o servidor de desenvolvimento:

```
npm run dev
```

4. Acessar a aplicação via navegador no endereço exibido pelo Vite.

6 Considerações Finais

A estrutura de diretórios adotada para o projeto *Seu Cantinho* — com separação clara entre backend, frontend, banco de dados e artefatos UML — favorece organização, manutenção e extensibilidade.

O mapeamento UML → código foi respeitado tanto na modelagem do schema relacional quanto na definição das interfaces TypeScript e controllers, permitindo rastrear facilmente onde cada entidade e caso de uso foi implementado.

Evoluções futuras podem incluir autenticação completa por papéis (admin, gestor, cliente), integração com gateways de pagamento reais e relatórios avançados para análise de ocupação e faturamento.